Programming Languages
Application and Interpretation



Version 2007-04-26

Shriram Krishnamurthi

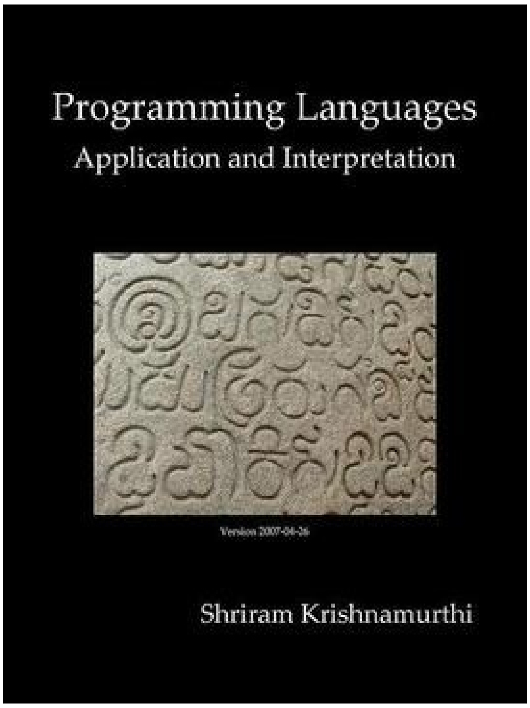
目錄

PLAI-cn

> 来源：lotuc/PLAI-cn

翻译准侧

1. 尽可能忠于原文（总原则）
2. 代码中\全部保留，在同一行最后分号注释加上中文翻译，比如 `<answer-type-take-1> ::= ;返回值类型，第一次尝试`
3. 术语翻译，可以在第一次出现/定义的地方后附括号，括号内标注英文，比如 闭包（Closure）
4. Excise和Think now统一翻译成练习和思考题
5. 注释也请翻译，根据实际情况，直接放在文中相应位置，或者放在本节最后处
6. 代码中出现的英文字符串，也在同一行后加中文注释，比如 `(error 'num+ "one argument was not a number") ;有一个参数不是数字`
7. md中代码一律使用Racket标记（不使用scheme，会导致中括号显示出问题）
8. 英文原文的斜体和粗体字，统一对应为中文的粗体字



PLAI 第二版

Gitbook - PLAI-cn

zhihu - lotuc

1. The translation has not been checked by the original author
2. The correct, definitive version is at the original link.

译者

- lotuc
- MrMathematica

1 引言

1.1 我们的哲学

请参见Youtube视频。

1.2 本书的结构

与某些教科书不同，本书并没有采取自上而下的叙述方式，而是采用了对话发展的方式，有时也会回头描述讲过的话题。如同现实中的程序员，我们通常一步一步来构造程序。有时候我们的程序也会包括错误，这并不是因为我不知道该怎么写出正确的程序，而是因为这是帮助你学习的最好方式。错误会迫使你没法被动的学习，而是必须钻研：你永远也没法确信读到的材料就是真实的。

最终，你会得到正确的答案。短期来说，这种方式使人挫折，而且读者也没法将本书当做参考书来使用（你没法打开书，翻到随便一页，就认为其中的内容是正确的）。但是，挫败感是学习的一个部分。我不觉得有好方法绕开它。

在书中你会遇到

练习

> 这是练习。请做题。

这和传统教材中的练习题一样，需要你独立完成。如果你确实在某个课程中使用本教材，有可能这就是课后作业。但是本书也包含这种：

思考题

> 这是思考题，你看到了吗？

当你看到思考题的时候，请停下来。阅读、思考，形成答案之后再继续。这是因为思考题本质上就是练习题，唯一的区别是后文会给出其答案，或者你可以通过运行程序自行得到答案。如果你不加思考的继续阅读，那么你就会读到答案（或者，如果答案是可以通过运行程序获得的情况下，完全忽略答案）。这样做既没有测试你的知识水平，也无法锻炼你的思维能力。换一种说法，思考题是鼓励你积极学习的一部分。

1.3 本书使用的语言

本书使用的主要语言是Racket。然而，跟很多操作系统一样，Racket支持很多编程语言，所以你必须显式的告诉Racket你在使用什么语言进行编程。在Unix系统的shell脚本中你需要在脚本开头添加如下一行来指明语言：

```
#!/bin/sh
```

在脚本的头部，你可能会类似的指定：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" ...>
```

类似的，Racket需要你声明所使用的语言。Racket语言可能使用和Racket一样的括号语法，但是有不同的语义；或语义相同语法不同；或者有不同的语法和语义。因此每个Racket 程序以 `#lang <语言名字>` 开头。默认的语言为Racket（名字为racket）。【注释】这本书中我们几乎总是使用语言：

```
plai-typed
```

> 在DrRacket版本5.3中，打开“语言/Language”菜单，“选择语言/Choose Language”菜单项，然后选择“使用代码中指定的语言/Use the language declared in the source”。

使用该语言时，除非特别指明，请在程序的第一行添加（本书后面例子代码中请假定我们添加了该行）：

```
#lang plai-typed
```

*Typed PLAI*语言和传统Racket最主要的不同是它是静态类型的。它还给你提供了些有用的的东西（construct）：`define-type`、`type-case` 和 `test`。【注释】下面是它们的使用实例。创建新的数据类型：

> 它还提供了其他一些有用的命令，比如控制测试输出的命令等。请参考该语言的文档了解。在DrRacket版本5.3中，打开“帮助/Help”菜单，选择“帮助台/Help Desk”菜单项，然后在帮助台的搜索栏中输入“plai-typed”。

```
(define-type MisspelledAnimal
  [caml (humps : number)]
  [yacc (height : number)])
```

它做的事情类似于在Java中：创建抽象类 `MisspelledAnimal`，它有两个实体子类：`caml` 和 `yacc`，它们的构造参数分别为 `humps` 和 `height`。

该语言中，我们通过下面方式创建实例：

```
(caml 2)
(yacc 1.9)
```

如同其名字暗示的，`define-type` 会创建给定名字的数据类型。当我们把该数据类型的值绑定到变量时就需要用到其类型：

```
(define ma1 : MisspelledAnimal (caml 2))
(define ma2 : MisspelledAnimal (yacc 1.9))
```

事实上这里你并不需要显式的声明类型，因为Typed PLAI在很多情况下（包括这里）都能够推断出正确的数据类型。因此上面的代码可以写成：

```
(define ma1 (caml 2))
(define ma2 (yacc 1.9))
```

不过我们倾向于对类型进行显式的声明。这么做一方面是尊崇规则，另一方面当我们日后阅读代码时有助于理解。

类型的名字可以递归的使用，本书会经常使用这种方式（例如2.4节中）。

该语言为我们提供了模式匹配功能，例如这个函数体：

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (type-case MisspelledAnimal ma
    [caml (humps) (>= humps 2)]
    [yacc (height) (> height 2.1)]))
```

在表达式 `(>= humps 2)` 中，`humps` 被绑定为 `caml` 实例的构造时所用到的参数。

最后，你应该编写测试案例，理想情况下，应该在开始定义函数之前写。当然在定义函数之后也需要写，以防代码被意外修改。

```
(test (good? ma1) #t)
(test (good? ma2) #f)
```

当你运行上面的代码时，语言会告诉你两个测试都通过了。要了解更多请参阅文档。

这里有一点可能比较费解。在模式匹配中，匹配数据字段时我们使用了和数据定义时相同的名字，`humps`（和 `height`）。这是完全没有必要的，模式匹配是基于位置的而不是名字。因此我们完全可以使用其它名字：

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (type-case MisspelledAnimal ma
             [caml (h) (>= h 2)]
             [yacc (h) (> h 2.1)]))
```

因为每个h仅在其被引入的匹配分支中可见，所以上面的代码没有重名的问题。命名是请尊崇传统和可读性。通常来说，定义数据类型时可以使用长而描述性的名字；而定义类型子句时请使用简短的名字，因为日后这些名字会不断被用到。

我觉得很少有需要你会用到类型判断函数（如caml?），不过你可以用。数据类型定义时还会生成字段提取函数，例如`caml-humps`。有时候，直接使用字段提取函数会比使用模式匹配更简单。当然一般来说还是模式匹配更好用，就如刚才的`good?`所示。不过为了完整，我们实现如下：

```
(define (good? [ma : MisspelledAnimal]) : boolean
  (cond
    [(caml? ma) (>= caml-humps ma) 2]
    [(yacc? ma) (> (yacc-height ma) 2.1)]))
```

思考题

> 如果给函数传入了错误的数据类型会发生什么？比如传给caml构造器一个字符串？或者传给前述两个版本的`good?`函数一个数？

2 本书有关语法解析的一切

语法解析（parsing）是将输入字符流转换成结构化内部表示的过程。常见的内部表示是树，程序可以递归地处理树这种数据结构。例如，给定输入流：

```
23 + 5 - 6
```

我们可以将其转换成根节点为加法，左边节点表示数`23`，右边节点是用树表示`5-6`的树。语法解析器（**parser**）是用于实现这种转换的程序。

语法解析本身是个比较复杂，且由于歧义的存在，还远没有被解决的问题。例如上面的例子，你还可以将其转换成根节点为减法，子树为加法的树。我们还需要考虑加法操作符的是否符合交换性（左右参数是否能互换）等问题。要解析功能完整的语言（暂且不提自然语言），要考虑的问题只会更多更复杂。

2.1 轻量级的，内建的语法解析器的前半部分

这些问题使得语法解析本身适合当作单独的主题来讲，也确实有很多书本、课程和工具专注于该方面。从我们的角度来说，语法解析是种令人分心的东西，因为我们想学习的是编程语言的除去语法解析的各个部分。因此，我们使用Racket一个有用的功能来将输入流转换成树：`read`。`read`和该语言的括号语法形式紧密关联，它将括号形式的字符流转换成内部树形式。例如，运行`(read)`然后输入——

```
(+ 23 (- 5 6))
```

——会产出一链表，其第一个元素是符号`'+`，第二个元素是数`23`，第三个元素是链表；该链表其第一个元素是符号`'-`，第二个元素是数`5`，第三个元素是数`6`。

2.2 快捷方式

你应该知道，程序都会需要详尽的测试，而每次测试都需要手工输入会很麻烦。幸运的是，你可能猜得到，括号表达式可以在Racket中用引号来表达，也就是你刚才看到的`'<expr>`形式——其效果和运行`(read)`然后输入`<expr>`一样。

2.3 语法解析得到的类型

事实上，我刚才的描述并不准确。之前说`(read)`会返回链表等类型。在Racket中确实如此，但在Typed PLAI中，事情稍有不同，`(read)`返回值类型为`s-expression`（符号表达式的简写）。

```
> (read)
- s-expression
[type in (+ 23 (- 5 6))]
'(+ 23 (- 5 6))
```

Racket包含了强大的s-expression系统，其语法还甚至可以表达带循环的结构。不过我们只会用到其中的一部分。

在静态类型的语言中，s-expression被认为是和其他类型（例如数、链表）都不同的数据。在计算机内部，s-expression是一种递归数据类型，其基本构造是原子值——例如数、字符串、符号，组合形式可以是表、向量等。因此，原子值（数、字符串、符号等）即是其自由类型，也是一种s-expression。这就造成了输入的歧义，我们后文讨论。

Typed PLAI采取一种简单的方式来处理这种歧义：当直接输入时，原子结构就是它本身的类型；当输入为大结构的一部分时——包括read或者引用——它们就是s-expression类型。你可以通过类型转换将其转换为基本类型。例如：

```
> '+
- symbol
'+
> (define l '(+ 1 2))
> l
- s-expression
'(+ 1 2)
> (first l)
. typecheck failed: (listof '_a) vs s-expression in:
  first
  (quote (+ 1 2))
  l
  first
> (define f (first (s-exp->list l)))
> f
- s-expression
'+
```

这方面和Java程序的类型转换类似。我们后文再学习类型转换。

请注意，表结构的第一个元素的类型并不是符号：表形式的s-expression是由s-expressions组成的表。因此，

```
> (symbol->string f)
. typecheck failed: symbol vs s-expression in:
  symbol->string
  f
  symbol->string
  f
  first
  (first (s-exp->list l))
  s-exp->list
```

类型转换：

```
> (symbol->string (s-exp->symbol f))
- string
"+"
```

必须对s-expressions进行类型转换确实是个麻烦事，但是某种程度的麻烦是不可避免的：因为我们的目的是把没有类型的输入，通过严谨的类型分析，转化为有类型的。所以有些关于输入的假设必须明文给出。

好在我们只在语法解析中使用s-expressions，而我们的目的是尽快处理完语法解析！所以，这一点只会帮助我们尽快摆脱语法解析。

2.4 完整的语法解析器

原则上`read`就是完整的语法解析器。不过其输出过于一般化：结构体中并不包含其意向的注释信息。所以我们倾向于使用更具体的表达方式，类似于前文中“表达加法”和“表达数”的那种。

首先，我们必须引入一种数据类型来表示这类关系。后文（3.1节）会详细讨论为啥采用这种数据类型，还有我们如何得出该数据类型。现在请先假设它是给定的：

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

现在我们需要能将s-expression解析成该数据类型的函数。这就是语法解析器的另一半：

```
(define (parse [s : s-expression])
  (cond
    [(s-exp-number? s) (numC (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusC (parse (second sl)) (parse (third sl)))]
         [(*) (multC (parse (second sl)) (parse (third sl)))]
         [else (error 'parse "invalid list input")]))]  ;无效的表输入
    [else (error 'parse "invalid input")]))  ;无效的输入
```

简单运行如下：

```
> (parse '(+ (* 1 2) (+ 2 3)))
- ArithC
(plusC
 (multC (numC 1) (numC 2))
 (plusC (numC 2) (numC 3)))
```

恭喜！你完成了首个程序的表示。从今往后我们就只需要处理用递归的树结构表示的程序了，再也不用担心各种不同的语法，还有如何把语法转换为树形结构了。我们终于可以开始学习编程语言了！

练习

> 如果传给语法解析器的参数忘了加引号，后果是啥？为什么？

2.5 尾声

Racket的语法继承自Scheme和Lisp，不乏争议。不过请观察它给我们带来的深层次好处：对传统语法进行解析会很复杂，而解析这种语法则简单明了，不管是从字符流到s-expressions的解析，还是从s-expressions进一步到语法树的解析。

这种语法的好处就是其多用途性。需要的代码少，而且可以方便的插入各种应用场景。所以很多基于Lisp的语言其语义各不相同，但都保留了历史继承而来的这种语法。

当然，我们也可以采用XML，它更好用；或者JSON，它和s-expression有着本质的不同！

3 解释器初窥

现在有了程序的表示方法，我们有很多方式可以用来操纵它们。我们可能想把程序打印的漂亮点（pretty-print），将其转换成其它格式的代码（编译），查看其是否符合特定属性（校验），等等。现在，我们专注于考虑得到其对应的值——计算（evaluation）——将程序规约成值。

让我们来为我们的算术语言写个解释器形式的求值器。选择算术运算是出于下面三个主要原因：（a）你已经知道怎么计算加减乘除了，我们可以专注于其实现；（b）基本上每门语言都会包含算术运算，所以我们可以从它开始进行语言的扩展；（c）该问题大小合适，足以展示我们要学习的很多要点。

3.1 算术表达式的表示

我们首先需要统一算术表达式的表示法。我们只打算支持两个运算符——加法和乘法——以及基本的数。需要一种东西来表达算术表达式。算是表达式的嵌套规则是啥呢？表达式可以任意地嵌套。

思考题

> 为什么我们不把除法也包括进来呢？这么做对前文总结会产生什么影响？

这里不包括除法的原因是，我们暂时不打算讨论什么表达式是合法的。显然1除以2是合法的，但是1除以0就有争议了。1除以（1减去1）就更有争议了。目前我们无需陷入这种矛盾，以后再讨论。

于是我们可以使用如下的表达式：

```
(define-type ArithC  ;具体算术
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

3.2 写个解释器

下面开始写该算术语言的解释器。首先我们考虑一下该解释器的类型：它的输入显然是 `ArithC` 值，返回值的类型呢？当然是数啦。即我们的解释器是输入为 `ArithC` 输出为数的函数。

练习

> 为该解释器写一些测试案例。

由于输入类型是递归定义的数据类型，很自然的解释器也应该递归地处理输入。程序模板如下：【注释】

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
             [numC (n) n]
             [plusC (l r) ...]
             [multC (l r) ...]))
```

> 《程序设计方法》一书（又译《如何设计程序》）详细介绍了模板这一概念。

你很可能想当然的直接写出如下的代码：

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
             [numC (n) n]
             [plusC (l r) (+ l r)]
             [multC (l r) (* l r)]))
```

思考题

> 你能找到其中的错误吗？

首先，我们先补充模板代码：

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
             [numC (n) n]
             [plusC (l r) ... (interp l) ... (interp r) ...]
             [multC (l r) ... (interp l) ... (interp r) ...]))
```

填充必要部分得到解释器：

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
             [numC (n) n]
             [plusC (l r) (+ (interp l) (interp r))]
             [multC (l r) (* (interp l) (interp r))]))
```

这样，我们就完成了第一个解释器！我知道有点虎头蛇尾，但是我保证，它会变得越来越复杂。

3.3 你注意到了吗？

有件事情我没和你讲清楚：

思考题

> 在这个语言中，加法和乘法的“意义”是啥？

太抽象了，不是吗？让我们把它变得更具体一些。计算机中有很多种不同的加法：

- 首先，有很多种不同的数：固定长度（例如，32位）整数，带符号固定长度（例如，31位外加1个符号位）整数，任意精度整数；在有些语言中，有理数；各种不同格式的固定位数浮点数；在有些语言中，复数；如此等等。在确定数类型之后，加法可能只支持其中的一部分组合。
- 其次，某些语言支持某些（其他）数据类型的加法，比如矩阵加法。
- 再次，某些语言支持字符串“相加”。这里引号表示我们并没有进行数学上相加的操作，而是用语法上用+符号表示操作。有的语言用这表示字符串拼接；也有语言在这种情况下返回数（比如把字符串所表示的数相加）。

这些都是加法所代表的不同含义。语义是把语法（例如+）映射到含义（例如，以上列举的部分或者所有）。

于是游戏来了：以下哪些是相同的？

- 1 + 2
- 1 + 2
- ’1’ + ’2’
- ’1’ + ’2’

回到之前的问题，我们用的语义是啥？我们直接使用了Racket所提供的语义，因为程序直接把+映射到了Racket的+上。其实这也不一定是对的：比如说，如果Racket的 `+` 也支持字符串，那么我们这里提供的操作就限制+只能用在数上（事实上Racket的 `+` 并不支持字符串）。

如果我们想要不同的语义，需要显式的实现出来。

练习

> 需要哪些修改，这里的加法能支持带符号32位数的算术?

一般来说，我们需要避免简单的借用宿主语言的语义。后面我们还会讨论这个话题。

3.4 扩展此语言

我们选择的第一个语言功能非常有限，于是有很多种方式可以将其扩展。有的扩展，比如添加数据结构和函数，就必须要增加解释器所支持的数据类型（假设我们并不打算采用哥德尔计数法）。其他的扩展，比如增加更多算术操作，就不必修改核心语言及其解释器。我们下一章就讨论此问题。

4 初试去语法糖

我们从非常斯巴达式的算术语言开始。下一步我们来看看，在现有语言框架下怎么支持更多算术操作。我们只加2种，以做示范。

4.1 扩展：添加双目减法操作

首先，我们来添加减法。由于我们的语言已经包含了数、加法和乘法，用这些操作足以定义减法了：

```
a - b = a + -1 × b
```

好的，这很简单！但是我们要怎样将它变成可运行的代码呢。首先，我们面临一个决定，将减法操作符放在哪？将其像其它两个操作符一样处理，在现有的ArithC数据类型中添加一条规则？这种想法看上去很自然，也很诱人。

思考题

> 修改ArithC这种做法有什么不好的地方呢？

这会导致几个问题。首先，显然地，我们将需要修改所有处理ArithC的代码。就目前而言，还很简单，只涉及到了我们的解释器。但是如果在更为复杂的语言实现中，这会是个问题。其次，要添加的结构是可以用已实现的语法结构定义的，去修改已有数据结构的方式让人觉得代码不够模块化。最后一点，也是最微妙的一点，修改ArithC这种行为有概念上的错误。因为ArithC描述的是我们语言的核心部分。而减法（和其他类似添加特性）是用户交互的部分，属于表层语言。明智的做法是，将不同类型的概念放到不同的数据类型中，而不是把它们硬塞到一起。有时候这么做看上去有点笨拙，不过长远来看，它会让我们的程序更易于阅读易于维护。此外，你可能会将不同的功能扩展放在不同的层次上，这么做（将核心语法和表层语法区分开）正有利于这么做。

因此，我们尝试定义新的数据类型来反应我们的表层语言语法结构：

```
(define-type ArithS  ;表层算术
  [numS (n : number)]
  [plusS (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [multS (l : ArithS) (r : ArithS)])
```

它看起来和ArithC基本相同，遵从了相似的递归结构，唯一的区别就是加了一个子句。

数据类型定了，接下来需要做两件事。第一是要修改语法解析器，让其返回ArithS类型数据（而不是ArithC类型）。第二是要实现去语法糖（desugar）函数，它需要能把 `ArithS` 值转换成 `ArithC` 值。

先来实现去语法糖函数简单的部分：

```
<desugar> ::=  ;去语法糖

    (define (desugar [as : ArithS]) : ArithC
      (type-case ArithS as
        [numS (n) (numC n)]
        [plusS (l r) (plusC (desugar l)
                            (desugar r))]
        [multS (l r) (multC (desugar l)
                            (desugar r))]
        <bminusS-case>))  ;二元减法子句
```

把数学描述转化为代码：

```
<bminusS-case> ::=  ;二元减法子句

    [bminusS (l r) (plusC (desugar l)
                          (multC (numC -1) (desugar r)))]
```

思考题

> 常见错误是忘了递归地对 `l` 和 `r` 进行desugar操作。忘了会发生什么？请自行尝试。

4.2 扩展：取负数操作

让我们来考虑另一种更有意思的扩展，取负数操作（unary negation）。这使得你需要对语法解析器进行一定修整，当读到 `-` 符号时，需要往前读以判断它是减法还是取负操作。但这不是有意思的部分！

取负数操作可以有几种去语法糖的方法。很自然的我们会想到：

```
-b = 0 - b
```

继续完成减法的去语法糖操作，我们得到：

```
-b = 0 + -1 × b
```

思考题

> 你觉得这两种中哪个更好呢？为什么？

大家可能希望使用第一种方式，因为它看起来更为简单。假设我们扩展了 `ArithS` 数据类型，添加取负数的表示法：

```
[uminusS (e : ArithS)]  ;一元减法表达式
```

对应去语法糖的实现也很直接：

```
[(uminusS (e) (desugar (bminusS (numS 0) e)))]
```

检查看看有没有类型错误。`e` 是 `ArithS` 类型，所以它可以被当作参数传递给 `bminusS` 来进行去语法糖操作。所以这里要做的不是对e去语法糖，而是将其直接嵌入到生成的表达式中。在去语法糖的工具中，这种直接将某个输入项嵌入到另一个项中，然后递归调用去语法糖函数的做法很常见，被称之为宏（**macro**）。（在我们这个例子中，“宏”是 `umiunsS` 的定义。）

然而该定义存在两个问题：

1. 第一个问题是，该递归是生成的（**generative**），这需要我们得对其进行特别关注。【注释】我们可能会希望使用下面这种方式来重写它：

```
[uminusS (e) (bminusS (numS 0) (desugar e))]
```

它确实消除了生成性（generativity）。

> 如果你没听过生成递归，可以阅读《程序设计方法》（又译《如何设计程序》）一书第五部分。简单来说在生成递归中，子问题是输入的计算结果，而不是输入的子成分。我们这个例子还是很简单的，这里的“计算”就是 `bminusS` 构造函数。

思考题

> 很不幸的是，上面的转换有问题，试着找出问题吧。找不出的话，运行一下试试。

第二个问题是，它依赖于 `bminusS` 的意义；如果 `bminusS` 的意义发生变化， `uminusS` 的意义也就发生了变化，即使我们并没打算改变 `uminusS` 的意义。作为对比，另一种更鲁棒的做法是，定义函数，其输入是两个项，输出是第一个项加上 `-1` 乘以第二个项的表示法，然后用该函数来定义 `uminusS` 和 `bminusS` 。

你可能会说，减法的意义不可能发生改变，这么做有啥意义呢？事情并不总是这样的。确实减法的意义不太可能改变；但是另一方面，它的实现可能会改变。例如，开发者决定为减法操作打印日志。采用前一种做法（宏展开），所有取负数操作就也会打出日志；而采用后一种做法就不会。

很幸运，这个例子我们还有更简单的选择：

```
-b = -1 × b
```

这种展开方式完全可行，而且还是结构递归。我们花这些篇幅讨论各种不同展开方式的原因是，告诉你各种选择和其带来的问题，毕竟现实中你不会总是那么幸运。

5 添加函数

下面尝试将其变成真正的语言。比如说可以添加诸如条件语句这个特性，但是一个语言要真正变得有意思，它需要函数或者某种等价于函数的东西。所以我们就直接来添加函数好了。

练习

> 给语言添加条件语句。你可以选择添加布尔类型，或者方便起见，你的条件语句可以将0视作false，其他值视作 true。

想象一下，我们要构造一个类似于DrRacket的系统。程序员在定义（definitions）窗口中定义函数，然后在交互（interactions）窗口中使用它们。我们先假设函数只能在定义窗口定义；交互窗口中只能出现表达式（这些限制会随着内容的深入被解除）。按此假定，当运行程序时，函数已经被解析可供使用。所以，我们给解释器添加一个参数——函数定义的集合。

> 注意这里我们说的是函数的集合，也就是说，任何函数的定义中可以引用任意其它函数。这是我有意的设计。当你设计自己的语言时，记住注意考虑这一点。

5.1 定义函数的数据表示

简单起见，我们仅考虑只有一个参数的函数。下面是一些Racket函数的例子：

```
(define (double x) (+ x x))

(define (quadruple x) (double (double x)))

(define (const5 _) 5)
```

练习

> 如果函数可以带有多个参数呢？参数名之间有什么限制？

函数的定义包含哪些内容？它包含名字（上文中的`double`，`quadruple`，`const5`），我们将使用符号（symbol）类型表示（`'double`等）；形参（formal parameter，形式参数的简写）（例如`x`），也使用符号类型表示（`'x`）；最后还有函数体。我们后面会一步一步完善函数体的表示法，现阶段函数定义的数据类型如下：

```
<fundef> ::=  ;函数定义

    (define-type FunDefC
      [fdC (name : symbol) (arg : symbol) (body : ExprC)])
```

所以函数体是什么呢？显然它可以是算术表达式，且有时候应该可以使用 `ArithC` 语言来表示：例如，函数 `const5` 的函数体可以使用 `(numC 5)` 表示。但是要表示 `double` 函数的函数体需要更多东西：不仅需要加法（我们已经定义了），还需要“x”。你可能会称它变量（variable），但是现在我们不使用该术语，我们叫它标识符（identifier）。

> 后文我还会进一步解释这两个命名。

思考题

> 还有别的吗？

最后，我们看看 `quadruple` 的函数体，它包含另一种结构：函数调用（application）。要特别注意函数定义和调用的区别。函数定义描述了函数是什么，而调用则是对函数的使用。里面一层的 `double` 函数调用使用的实参（actual parameter，实际参数的简写）是 `x` ；外面的那层的 `double` 调用使用的参数是 `(double x)` 。可以看到，参数可以是任意表达式。

下面我们尝试把上面所有的东西糅合到一个数据类型中。显然我们需要扩展已有的语法（因为我们还想保留算术运算）。我们给新的数据类型一个新名字以示区别：

```
<exprC> ::=  ;表达式

    (define-type ExprC
      [numC (n : number)]
      <idC-def>  ;标识符的定义
      <app-def>  ;调用的定义
      [plusC (l : ExprC) (r : ExprC)]
      [multC (l : ExprC) (r : ExprC)])
```

标识符与形参关系紧密。当调用函数时，我们传给它某个值，从效果来说是将函数体中出现的形参实例——所有同名标识符——替换为该值。【注释】为了简化这个搜索——替换过程，不妨使用与形参相同的数据类型来表示标识符。形参的数据类型已经定好了，于是：

```
<idC-def> ::=  ;标识符的定义

    [idC (s : symbol)]
```

> 这里我们忽略了几个问题：“值”是什么？何时替换？后文会继续讨论。

最后，函数调用。它包含两个部分：函数名和（实际）参数。上面已经说过参数可以为任意表达式（包括标识符和函数调用）。至于函数名，让其和函数定义中的函数名类型一致符合直觉，就这样做吧：

```
<app-def> ::=  ;调用的定义

    [appC (fun : symbol) (arg : ExprC)]
```

该定义简单明了，函数名指明要调用哪个函数，然后后面提供函数调用所需参数。

有了定义，看看之前的三个函数该怎么表示：

- `(fdC 'double 'x (plusC (idC 'x) (idC 'x)))`
- `(fdC 'quadruple 'x (appC 'double (appC 'double (idC 'x))))`
- `(fdC 'const5 'x (numC 5))`

下面还需要选择函数定义集合的表示法。使用链表类型就蛮方便。

> 小心！你有没有注意到，之前我们说这是函数定义的集合，然而实现却选用了链表。也就是说，我们用有序的数据结构去表达无序的数据。那么，测试时，至少我们应该试用各种不同顺序的函数定义，以确保我们没有不小心引入了（影响结果的）顺序。

5.2 开始实现解释器

于是我们可以开始实现解释器了。首先考虑解释器的输入是什么。之前，只需要传入一个表达式即可，现在它还需要传入函数定义的表。

```
<interp> ::=  ;解释器

    (define (interp [e : ExprC] [fds : (listof FunDefC)]) : number
      <interp-body>)  ;解释器主体
```

稍微回顾一下我们前面实现的解释器（第三章）。遇到数，显然还是直接返回该数作为结果；遇到加法和乘法，还是应该一样递归的求值。递归时该用什么作函数定义呢？由于求值过程中，既不需要添加也不需要移除函数定义，即函数定义集合保持不变，在递归时函数定义应该原封不动的往下传递。

```
<interp-body> ::=  ;解释器主体

    (type-case ExprC e
      [numC (n) n]
      <idC-interp-case>  ;解释之标识符子句
      <appC-interp-case>  ;解释之调用子句
      [plusC (l r) (+ (interp l fds) (interp r fds))]
      [multC (l r) (* (interp l fds) (interp r fds))])
```

接下来实现函数调用。首先我们需要从函数定义中寻找对应的函数定义，我们可以假设如下的帮助函数可以实现此功能：

```
; get-fundef : symbol * (listof FunDefC) -> FunDefC
```

假设我们已经找到了函数的定义，下一步要对其函数体求值。还记得之前说过函数调用该怎么求值？搜索标识符并将其替换为实际参数。这个搜索替换过程足够重要，值得花一小节讨论，5.4节我们再回过来实现解释器。

5.3 替换

替换是将一个表达式（这里是函数体）中某个名字（这里是形参）替换成另一个表达式（这里是实参）的过程。首先确定其类型：

```
; subst : ExprC * symbol * ExprC -> ExprC
```

将参数名起的有意义些：

```
<subst> ::=  ;替换

    (define (subst [what : ExprC] [for : symbol] [in : ExprC]) : ExprC
      <subst-body>)  ;替换函数的主体
```

在 `in` 表达式中，将 `for` 替换成 `what` 。

思考题

> 考虑之前几个示例函数的函数体，将参数 `x` 替换为 `3` 的结果是什么？

对于 `double` 函数来说，结果为 `(+ 3 3)` ；对于 `quadruple` ，结果为 `(double (double 3))` ；对于 `const5` ，结果就为 `5` （函数体中没有出现 `x` 所以也没有替换）。

> 对于 `double` 一个常见的错误是将其替换成 `(define (double x) (+ 3 3))` 。替换发生在函数调用时，此时只需要函数体就可以了。函数定义头部的作用是找到函数，还有给出参数的名称；但是计算其值时只需要函数体。如果用整个函数定义进行替换，试试看你会得到哪种类型错误。

这个例子几乎涵盖了所有情况。如果是数的话，无需替换任何东西；如果是标识符，例子没有覆盖标识符不同的情况，你也能想到该怎么做：保留之；其它情况，递归的替换各子表达式。

在开始写代码之前，还有一种重要情况要考虑一下。假设我们要替换的标识符恰巧是某个函数名称，该怎么处理呢？

思考题

> 该怎么处理呢？

对于这个问题，有多种处理方法。一种方案是从设计上来来考虑：函数名有其自己的“世界”，它和程序中其它标识符都不同。某些语言（例如C和Common Lisp，尽管它们的做法也略有不同）采取这种策略，根据标识符使用的位置将其解析到不同的命名空间。而其他一些语言则不做这样的区分。我们很快会研究这么一种语言（后文）。

现在，我们从务实的角度来处理这个问题。由于这里表达式求值结果是数，这就要问函数名能求值成数不。但是，数不能命名函数，只有符号能。所以进行这种替换是没有意义的，函数名和要替换的符号没有关系。（比如，某个函数的参数可以叫 `x` ，其函数体中又可以调用另一个名为 `x` 的函数，两者被区别处理。）

决定做完了，是时候写代码了：

```
<subst-body> ::=  ;替换函数的主体

    (type-case ExprC in
      [numC (n) in]
      [idC (s) (cond
                 [(symbol=? s for) what]
                 [else in])]
      [appC (f a) (appC f (subst what for a))]
      [plusC (l r) (plusC (subst what for l)
                          (subst what for r))]
      [multC (l r) (multC (subst what for l)
                          (subst what for r))])
```

练习

> 请注意，在numC子句，解释器返回`n`，而替换函数返回`in`（即原始表达式，在这个位置等价于`(numC n)`）。为什么？

5.4 继续实现解释器

搞定了替换的实现（我们这么认为），我们来完成解释器。替换这步干了很多事，好在函数调用的很多细节都在其中完成了。很自然的想法是：

```
<appC-interp-case-take-1> ::=  ;解释之调用子句，第一次尝试

    [appC (f a) (local ([define fd (get-fundef f fds)])
                  (subst a
                         (fdC-arg fd)
                         (fdC-body fd)))]
```

但是这是错的。

思考题

> 看出错在哪里了吗？

从类型角度考察，解释器的函数返回类型是什么？数。替换函数的返回类型呢？表达式。比如说，在替换`double`的函数体是，可能得到的结果是`(+ 5 5)`的表达形式。这并不是解释器的合法返回值。需要进一步对其求值。所以应该这么做：

```
<appC-interp-case> ::=  ;解释之调用子句

    [appC (f a) (local ([define fd (get-fundef f fds)])
                  (interp (subst a
                                 (fdC-arg fd)
                                 (fdC-body fd))
                          fds))]
```

好了，还剩下最后一个子句：标识符。这个能有多复杂呢？看上去标识符类似数那样简单！然而我们把它留到最后处理，这说明到了它的处理可能有点微妙或者说有点复杂。

思考题

> 请尝试一些例子，从而理解标识符该怎么处理。

假设 `double` 函数定义如下：

```
(define (double x) (+ x y))
```

我们把 `x` 替换成 `5` ，得到 `(+ 5 y)` 。没毛病，然而剩下的 `y` 该怎么替换呢？事实上从一开始我们就应该意识到这个 `double` 定义是错误的。标识符 `y` 被称为自由的（**free**），这是个负面的词。

换一种说法，解释器应该永远也遇不到标识符。在解释过程中，所有标识符应该都会被替换掉（被称为被绑定的，这是种正面的说法）。因此，当解释器直面标识符时，只能这么处理：

```
<idC-interp-case> ::=  ;解释之标识符子句

    [idC (_) (error 'interp "shouldn't get here")]  ;不应执行到这里
```

这样我们的解释器就完成了！

最后，为了完整，我们还需要实现 `get-fundef` ：

```
;获取函数定义
(define (get-fundef [n : symbol] [fds : (listof FunDefC)]) : FunDefC
  (cond
    [(empty? fds) (error 'get-fundef "reference to undefined function")]  ;引用未定义的
函数
    [(cons? fds) (cond
                   [(equal? n (fdC-name (first fds))) (first fds)]
                   [else (get-fundef n (rest fds))])]))
```

5.5 等等，还没完呢！

之前 `subst` 的类型我们说是：

```
; subst : ExprC * symbol * ExprC -> ExprC
```

简单起见我们这里用表面语法描述问题，假设我们在解释 `(double (+ 1 2))` 。它会将所有 `x` 都替换为 `(+ 1 2)` ，于是解释器得到表达式 `(+ (+ 1 2) (+ 1 2))` 。这是我们想要的吗？

在学习代数时，可能你的老师不是这么教你的：首先应该将参数规约成其结果（在这个例子中就是 `3` ），然后将参数替换为这个结果。这么说，替换的类型就应该是：

```
; subst : number * symbol * ExprC -> ExprC
```

请注意，我们不能直接把数放入表达式，而是必须先把数放入到numC调用中。所以，一种可行的做法是，`subst`函数可以把第一个参数用numC包装起来然后调用辅助函数。（事实上，现有的`subst`函数就可以是这个辅助函数：它接收的第一个参数的类型是ExprC，那么当传给它的数据是numC类型时显然没问题。）

> 事实上，替换的实现还是不太对！这里的替换函数仅仅能处理我们的示例语言，过了就不行了。这给问题也很微妙，它被称为“名称捕获”。解决这个问题是复杂，巧妙和令人兴奋的智力工作。不过这里我不打算往这个方向发展。所以本书中我们绕过此问题。不过如果你对此感兴趣，请阅读**lambda**演算方面的书籍，它们会提供帮助正确地实现替换。

练习

> 修改解释器，用答案而不是表达式替换标识符。

我们这里遇到的问题正是程序语言中一个基本设计抉择。如果在替换前就把参数的值求好，这被称为及早（eager）求值；对应的推迟求值被称为惰性（lazy）求值——它本身又有几种不同变化。我们这里倾向于使用及早求值语义，因为大部分主流语言都采取此方式。后面也会再介绍惰性求值的语义和后果。

6 从替换模型到环境模型

尽管我们已经实现了函数，你可能对其不太满意。处理标识符时，直观上应该是“找到它绑定的值”。但是我们不仅没这样做，还在遇到标识符时直接抛出错误！这么做也没错，但感觉怪怪的。更重要的是，编写解释器是为了让其理解并解释我们的语言，而该解释器现在看来并没有达成我们的意愿。

使用替换模型的另一个问题是，它需要遍历源程序的次数。理想的做法是，只访问程序中被实际求值的部分，并且，仅当必要时才这么做。替换模型必须遍历程序的所有部分——比如说，条件分支中不执行的分支——而且还需要遍历程序两遍，一遍替换，一遍解释。

练习

> 替换模型会影响程序运行的时间复杂性吗？

替换模型还有个问题，它的结构受限于源代码的存储。当然，我们的解释器需要源代码，对其解释。但是其他实现方式——比如说编译器——并不需要存贮源代码。【注释】采取更一般的策略，不依赖于具体实现方式显然更为合理。

> 编译器可能也需要存储某些源代码信息，以实现其他功能。比如说，报告运行时错误时，或者进行即时编译（JIT）。

6.1 介绍环境模型

直觉告诉我们，解决第一个问题的方法是，解释器可以在某种形式的字典里面“寻找”标识符；解决第二个问题的方法是，延迟替换。幸运的是，这两点结合起来还能解决第三个问题。字典里面记录的是将要进行的替换，而并不对原始程序进行修改。因为记录下了将要进行的替换，而并非直接替换，我们可以延迟进行替换步骤。记录替换内容的数据结构被称为环境（environment）。使用环境模型避免了源代码级别的重写，并且和底层的计算机表示法很好地对应。环境中的结合关系被称为绑定（binding）。

注意，这里我们要修改的是编程语言的实现策略，而不是修改语言本身。因此用于表示程序的数据结构，还有解释器执行的结果都不应发生改变。所以，我们可以将之前那个解释器当作我们这次要写的解释器的“参考实现”，两者的结果应该一致。实际上，我们应该创建一个测试生成器，让它生成很多测给两个解释器来执行，并确保它们返回的结果都相同。理想情况下，我们应该证明两个解释器行为一致，事实上它是很好的高阶课题。

> 这里的“一致”到底是什么意思呢？特别地，当程序运行报错时呢？

首先，我们来定义环境的数据结构。环境是将名字与什么的绑定的表？

思考题

> 这里定义数据结构时，很自然的问题就是，环境中将名字映射成了什么东西。但是我们可以问更好更基本的问题，我们如何得出这个很自然问题的答案？

记住我们这里引入环境是为了推迟替换过程。因此，答案在替换中。我们在上一章最后一节中讨论过，我们希望直接将名字替换为计算结果，即对应于函数的及早求值策略。因此同样的，环境中应该将名字映射为求值结果。

```
(define-type Binding
  [bind (name : symbol) (val : number)])

(define-type-alias Env (listof Binding))
(define mt-env empty)
(define extend-env cons)
```

6.2 环境模型解释器

现在可以实现解释器了。除最简单的分支情况外，其它代码均需要重新考虑：

```
<*> ::=

    (define (interp [expr : ExprC] [env : Env] [fds : (listof FunDefC)]) : number
      (type-case ExprC expr
        [numC (n) n]
        <idC-case>  ;标识符子句
        <appC-case>  ;调用子句
        <plusC/multC-case>))  ;加法乘法子句
```

算术操作是最好写的。回忆一下，递归中未涉及到新的替换，因此无需特别处理，环境不会发生改变：

```
<plusC/multC-case> ::=  ;加法乘法子句

    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
```

接下来我们处理标识符。显然，现在遇到标识符不应该直接报错了。我们应该在当前环境中查找对应的值：

```
<idC-case> ::=  ;标识符子句

    [idC (n) (lookup n env)]
```

思考题

> 实现lookup函数。

最后，处理函数调用。注意到在替换模型的解释器中，唯一创建新替换的部分就是函数调用。因此这个地方会是需要创建绑定的地方。第一步，跟之前一样，提取函数定义：

```
<appC-case> ::=  ;调用子句

    [appC (f a) (local ([define fd (get-fundef f fds)])
                  <appC-interp>)]  ;调用的解释
```

之前，我们是先进行替换，然后解释。现在剔除掉替换这个步骤，我们首先记录下要替换的东西，然后直接进入解释步骤：

```
<appC-interp> ::=  ;调用的解释

    (interp (fdC-body fd)
            <appC-interp-bind-in-env>  ;调用的解释，环境绑定
            fds)
```

也就是说，函数定义部分保持不变；我们照旧解释函数的主体部分；不过解释过程要被放在新的环境中，该环境包含了函数形式参数的绑定。接下来定义绑定过程：

```
<appC-interp-bind-in-env-take-1> ::=  ;调用的解释，环境绑定，第一次尝试

    (extend-env (bind (fdC-arg fd)
                      (interp a env fds))
                env)
```

要绑定的是形参（之前被替换的也是形参）。绑定的值是函数参数解释求值的结果（因为我们采取及早求值语义）。这就需要扩充我们的环境。类型检查确保我们得到的代码是正确的。

最后加上lookup函数的实现，一切就都完成了：

```
(define (lookup [for : symbol] [env : Env]) : number
  (cond
    [(empty? env) (error 'lookup "name not found")]  ;找不到名称
    [else (cond
            [(symbol=? for (bind-name (first env)))
             (bind-val (first env))]
            [else (lookup for (rest env))])]))
```

请注意，查找自由标识符时依旧会报错，但是这一步从解释器中被剥离出来——解释器并无法判断某个标识符是否被绑定——由lookup函数根据当前环境来决定。

完成解释器后，当然需要测试以确保其正确性。下面这几个测试都通过了：

```
(test (interp (plusC (numC 10) (appC 'const5 (numC 10)))
              mt-env
              (list (fdC 'const5 '_ (numC 5))))
      15)

(test (interp (plusC (numC 10) (appC 'double (plusC (numC 1) (numC 2))))
              mt-env
              (list (fdC 'double 'x (plusC (idC 'x) (idC 'x)))))
      16)

(test (interp (plusC (numC 10) (appC 'quadruple (plusC (numC 1) (numC 2))))
              mt-env
              (list (fdC 'quadruple 'x (appC 'double (appC 'double (idC 'x))))
                    (fdC 'double 'x (plusC (idC 'x) (idC 'x)))))
      22)
```

所以，我们是已经完成任务了，对吧？

思考题

> 找找看bug在哪。

6.3 正确的进行延迟求值

考虑下面这个测试：

```
(interp (appC 'f1 (numC 3))
                  mt-env
                  (list (fdC 'f1 'x (appC 'f2 (numC 4)))
                        (fdC 'f2 'y (plusC (idC 'x) (idC 'y)))))
```

在我们的解释器中，它的结果为7。这正确吗？

将这个测试转换成Racket代码，两个定义加一个表达式：

```
(define (f1 x) (f2 4))
(define (f2 y) (+ x y))

(f1 3)
```

考虑其求值过程。`(f1 3)`将函数`f1`的函数体中`x`替换为3，于是下一步处理`(f2 4)`。但是注意到在函数`f2`中， 标识符`x`未绑定！当然Racket报错了。

事实上，我们的替换模型解释器也会报错！

为什么我们的替换模型会报错呢？这是因为，我们仅会在`f1`的函数体内将标识符`x`替换为数3的表示。【注释】（这是显而易见的事：`x`是`f1`的参数；如果其它函数的参数名碰巧也叫`x`，那也是个不同的`x`）。当我们计算`f2`时，其中`x`没有被替换过，因此报错了。

> “数3的表示”听上去是不是很罗嗦？以后这类情况我就直接说“3”了，不过请你理解这其中的区别。

那么我们环境模型的问题到底出在哪呢？请仔细观察，这一点很微妙。只有函数调用过程会改变环境，我们重点观察这一步。将形参替换成实参是通过扩展当前环境实现的。在我们的例子中，在处理 `f2` 函数体时，我们不仅要求它对 `f2` 的参数进行替换，还要它对当前所有环境中的参数（也就是调用 `f2` 的 `f1` 的参数）都进行替换。如果还有上一层，它的参数也会被替换。换一种说法，添加到环境中的绑定只增不减。

由于前面说过，环境模型是替换模型的替代实现策略——我们的语言意义不应该发生改变——唯一合理的做法是修改解释器。具体来说，我们不应该让解释过程携带所有历史绑定，而应为每个函数创建干净的环境，类似替换模型的做法。很容易实现：

```
<appC-interp-bind-in-env> ::=  ;调用的解释，环境绑定

    (extend-env (bind (fdC-arg fd)
                      (interp a env fds))
                mt-env)
```

到此，我们重现了替换模型解释器的行为。

> 对于需要报错的情况，测试该怎么写呢？请查阅test/exn的文档。

6.4 作用域

上面那个错误的环境模型解释器，所实现的语义被称为动态作用域（**dynamic scope**）。它意味着随着程序的执行，环境中的绑定不断增加。于是，某个标识符是否被绑定取决于程序的执行历史。这应该被视作程序语言设计的缺陷。它增加了所有相关工具的复杂度，如编译器、IDE，也使得其代码难于阅读维护。

与之对应的，替换模型，以及上面正确实现的环境模型，给我们带来的是词法作用域（**lexical scope**） 或称静态作用域（**static scope**）。这里“词法（lexical）”指的是 “通过源码即可确定”；“静态（static）”指的是“不需运行程序即可确定”，在这里，这两者指代的意义相同。当遇到标识符时，我们希望知道两件事：

1. 它是否被绑定了？
2. 如果被绑定，在何处被绑定的？

这里“何处被绑定”指的是当程序中某个名字在多处被绑定，当前这个名字对应于哪个绑定。换一种说法，那个绑定负责当前标识符的绑定？一般来说，在动态作用域的语言中，这种问题没有静态的答案；于是你的IDE没法提示你某个变量是在哪个地方被定义的（DrRacket可以通过画箭头的方式提示此类信息）。【注释】因此，随着命名空间变得更为复杂（比如引入对象、线程等概念），我们仍需努力维护静态作用域原则。

> 思考这个问题的另一种方法是，在动态作用域的语言中，所有变量的绑定位置都没法静态确定，它总是取决于动态的环境。换一种说法，这种信息毫无价值。

6.4.1 动态作用域到底有多糟糕

可能看到上述的例子，你会觉得这是小题大作。但是，请考虑这两件事：

1. 要真正理解动态作用域的程序，你必需阅读整个程序。不管你怎么将程序进行解耦成易于理解的小的部分，如果程序中有个自由变量呢。
2. 要理解绑定关系，其复杂度不仅涉及到程序的体积，还牵扯到控制流的复杂度。考虑使用了很多回调的交互式程序，你需要追踪整个调用过程来确定某个标识符的值的来源。

还不够有说服力？让我们把示例程序中的表达式替换为：

```
(if (moon-visible?)
    (f1 10)
    (f2 10))
```

假设 `moon-visible?` 函数在新月的夜晚其值为假，其他情况下为真。于是我们的程序应当在新月夜晚报错，未绑定变量，而在其他情况下返回某个值。

练习

> *在多云的夜晚呢？*

6.4.2 全局作用域

当我们深入思考很多语言中全局的定义时，事情会变得更加复杂。例如，一些版本的Scheme（典型的词法作用域语言）允许你写出这样的程序：

```
(define y 1)
(define (f x) (+ x y))
```

看上去好像函数 `f` 中的 `y` 来源很清晰，不过：

```
(define y 1)
(define (f x) (+ x y))
(define y 2)
```

这是合法的，且计算 `(f 10)` 时它返回12。你可能会想，那么取最后一个定义就对了！。但是：

```
(define y 1)
(define f (let ((z y)) (lambda (x) (+ x y z))))
(define y 2)
```

这时候，`z` 绑定的是第一个 `y` 的值，lambda函数内部的 `y` 被绑定为第二个 `y` 的值。【注释】事实上可以通过词法作用域解释这种行为，但是它让情况变得异常复杂，可能避免这样的重定义是一种比较好的选择。Racket正是这样做的（在Racket中这么操作会导致报错），

它能提供用户全局定义，同时又避免这类麻烦事。

> 很多“脚本”语言都有类似的问题。所以网上你会看到很多人搞不清楚某种语言到底是静态还是动态作用域的。其实很多情况下人们只是在比较函数内的行为（通常是静态作用域）还是全局的行为（通常是动态作用域）。请注意这一点。

6.5 暴露环境

如果是实现供别人使用的解释器，明智的选择是将环境隐藏起来，给用户提供的接口只接收一个表达式，外加一系列函数定义，然后我们在程序内部从空白的环境开始调用interp。这样即不用将实现细节暴露给用户，也不会由于用户提供错误的环境导致问题。然而，有些情况下，暴露环境参数也是有用的。比如如果语言希望默认绑定一系列值:比如说，将 `pi` 绑定到3.2（Indiana）。

7 任意位置的函数

Scheme语言报告修订版报告的概述（r6rs 概述，r5rs（中文））中指出如下的设计原则：

> 程序语言的设计不应该是特性的简单堆砌，而应消除语言的弱点和缺陷，使得剩下的特性显得必要。

这是个无须争辩的设计原则。（当然有些缺陷是迫不得已的，但是此原则迫使我们去认真思考引入这些缺陷的必要性，而不是把它们当作理所当然的。）下面我们试着遵从该原则来引入函数。

在第五章中我们引入函数时并没有特别指明函数定义所在的位置。可以说我们是按照理想化的DrRacket模型引入的函数，即将函数的定义和使用分离。下面我们使用Scheme的设计原则来重新思考一下这种设计的必要性。

为什么函数的定义不可以也是一种表达式呢？我们现在实现的算术语言中有个尴尬的问题："函数的定义表示的是什么值？"，在现有设计中没有很好的答案。对于真正的语言来说，计算结果当然不可能只有数，所以也没必要给我们的语言作出这种限制；跳出这个框框，便可以给出很好的回答："函数值"。让我们试试如何实现它。

将函数作为值，能用它做什么呢？显然，函数和数是不同类型的值，你不能对函数做加法运算。但是，有件它显然能做的事：传入参数调用它！因此我们应该允许函数值出现在函数调用那个位置。其行为，显然是调用该函数。因此，我们的语言中应该允许如下的表达式作为合法程序（这里使用方括号以方便阅读）：

```
(+ 2 ([define (f x) (* x 3)] 4))
```

计算它得到 `(+ 2 (* 4 3))`，也就是 `14`。（注意到没？这里使用了替换计算模型。）

7.1 函数作为表达式和值

首先在我们的核心语言中添加函数定义：

```
<expr-type> ::=  ;表达式类型

    (define-type ExprC
      [numC (n : number)]
      [idC (s : symbol)]
      <app-type>  ;调用类型
      [plusC (l : ExprC) (r : ExprC)]
      [multC (l : ExprC) (r : ExprC)]
      <fun-type>)  ;函数类型
```

现在，我们简单把函数定义复制到表达式语言中，以后需要的话还可以修改这一点。这样做我们现在可以复用已有的测试案例。

```
<fun-type-take-1> ::=  ;函数类型，第一次尝试

    [fdC (name : symbol) (arg : symbol) (body : ExprC)]
```

接下来确定函数调用是什么样的。函数的位置应该放什么呢？我们希望它可以是函数定义，而不是像之前那样只能是定义好的函数的名字。由于现在函数定义类型和其它表达式类型混在了一起，这里让函数的位置可以放任意表达式吧，但是需要记住我们其实只希望它为函数定义：

```
<app-type> ::=  ;调用类型

    [appC (fun : ExprC) (arg : ExprC)]
```

> 另一种可以考虑的做法是，把函数定义和其他类型的表达式区分开。也就是定义不同类型的表达式。我们在后文学习类型时会考虑这种做法。

有了这个定义后，我们不再需要通过名字查找函数了，所以我们的解释器也可以不用再传入函数定义链表。当然之后有需要我们还可以将预定义函数链表加回来，现在我们只探究即时函数——在函数调用处定义的函数。

接下来修改解释器 `interp` 。需要添加子句来处理函数定义，该部分代码大致会是这样：

```
  [fdC (n a b) expr]
```

思考题

> 解释器中添加了该语句会导致什么？

显然，这是爆炸性的改变：解释器不再总是返回数了，于是出现类型错误。

在之前解释器实现过程中，也不时的需要注意其返回值类型，但并没专门给其定义数据类型。现在是时候需要这么做了：

```
<answer-type-take-1> ::=  ;返回值类型，第一次尝试

    (define-type Value
      [numV (n : number)]
      [funV (name : symbol) (arg : symbol) (body : ExprC)])
```

我们使用后缀 `V` 表示值（value），即求值的结果。 `funV` 部分正对应 `fdC` ； `fdC` 为输入， `funV` 为输出。通过区分这两者类型，我们可以分别修正它们两个。

下面我们尝试使用该输出类型重写解释器，从类型开始：

```
<interp-hof> ::=  ;解释器，高阶函数

    (define (interp [expr : ExprC] [env : Env]) : Value
      (type-case ExprC expr
        <interp-body-hof>))  ;解释器主体，高阶函数
```

这就要求我们同样修改`Binding`和辅助函数`lookup`的类型。

练习

> 修改`Binding`和辅助函数`lookup`。

```
<interp-body-hof> ::=  ;解释器主体，高阶函数

    [numC (n) (numV n)]
    [idC (n) (lookup n env)]
    <app-case>  ;调用子句
    <plus/mult-case>  ;加法/乘法子句
    <fun-case>  ;函数子句
```

对于数，显然要使用新的返回值类型构造器对其包裹一下。对于标识符，一切不变。对于加法／乘法，需要进行简单的修改使其能正确的返回`Value`类型而不是简单的数：

```
<plus/mult-case> ::=  ;加法/乘法子句

    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) (num* (interp l env) (interp r env))]
```

辅助函数`num+`和`num*`我们以其中一个为例：

```
(define (num+ [l : Value] [r : Value]) : Value
  (cond
    [(and (numV? l) (numV? r))
     (numV (+ (numV-n l) (numV-n r)))]
    [else
     (error 'num+ "one argument was not a number")]))  ;有参数不是数
```

请留意，在实际做加法前，我们检查了参数的类型确定其为数。后面会有章节继续谈论类型这个主题。

还有两段代码要完成。先是函数定义。上面说过，函数值就是其类型的数据：

```
<fun-case-take-1> ::=  ;函数子句，第一次尝试

    [fdC (n a b) (funV n a b)]
```

最后剩下函数调用的代码。尽管我们不再需要从函数定义链表中查询函数定义，但是这里还是尽量保留之前函数调用的代码的结构：

```
<app-case-take-1> ::=  ;调用子句，第一次尝试

    [appC (f a) (local ([define fd f])
                  (interp (fdC-body fd)
                          (extend-env (bind (fdC-arg fd)
                                            (interp a env))
                                      mt-env)))]
```

在原来是lookup查找的地方，我们直接引用了 `f` 作为函数定义。注意由于在函数应该出现的位置事实上可能出现任何表达式，我们最好编码检测它是否实是函数。

思考题

> 这里“是”是什么意思呢？我们是要检查它是作为语法结构上的函数（即 `fdC` 构造），还是只是检查该表达式的计算结果是否是函数值（即 `funV` ）呢？这两种做法有什么区别？换一种说法，你能不能找出具体的例子来展示其区别？

我们面临选择：

1. 检查它在语法上是否是 `fdC` 构造，如果不是，抛出异常。
2. 对其进行求值，然后检查其返回值是否是函数，如果不是，抛出异常。

我们选择后一种做法，它会使得我们的语言更为灵活。即使我们人类不一定需要这么做，但对于程序来说，第二种选择可以处理更多情况，比如程序生成代码。并且我们也会用到这个功能，就在匿名之上的语法糖的讨论中。于是，修改函数调用部分代码得到：

```
<app-case-take-2> ::=  ;调用子句，第二次尝试

    [appC (f a) (local ([define fd (interp f env)])
                  (interp (funV-body fd)
                          (extend-env (bind (funV-arg fd)
                                            (interp a env))
                                      mt-env)))]
```

练习

> 修改代码实现两种不同方式的类型检查。

信不信由你，到此为止，一个可运行的解释器又完成了。最后我们照旧给出两个测试案例：

```
(test (interp (plusC (numC 10) (appC (fdC 'const5 '_ (numC 5)) (numC 10)))
              mt-env)
      (numV 15))

(test/exn (interp (appC (fdC 'f1 'x (appC (fdC 'f2 'y (plusC (idC 'x) (idC 'y)))
                                          (numC 4)))
                        (numC 3))
                  mt-env)
          "name not found")
```

7.2 什么？嵌套？

函数定义的函数体部分可以是任意表达式。而函数定义本身也是表达式。于是函数定义中可以包含…函数定义。例如：

```
<nested-fdC> ::=  ;嵌套的fdC

    (fdC 'f1 'x
         (fdC 'f2 'x
              (plusC (idC 'x) (idC 'x))))
```

对它求值还不是特别有意思：

```
(funV 'f1 'x (fdC 'f2 'x (plusC (idC 'x) (idC 'x))))
```

当时如果我们调用上面的函数：

```
<applied-nested-fdC> ::=  ;调用嵌套的fdC

    (appC <nested-fdC>
          (numC 4))
```

再求值，结果就有点意思了：

```
(funV 'f2 'x (plusC (idC 'x) (idC 'x)))
```

这个结果就好像外部函数的调用对内部的函数没有任何影响一样。那么，为什么应该是这样的呢？外部函数引入的参数被内部函数引入的同名参数覆盖（mask）了，因此遵从静态作用域（必须的）的规则，内部的参数应该覆盖外部参数。但是，我们看看下面这个程序：

```
(appC (fdC 'f1 'x
           (fdC 'f2 'y
                (plusC (idC 'x) (idC 'y))))
      (numC 4))
```

求值得到：

```
(funV 'f2 'y (plusC (idC 'x) (idC 'y)))
```

嗯，有点意思。

思考题

> 想想有意思的点在哪？

为了看看到底有意思在哪，我们调用一下该函数：

```
(appC (appC (fdC 'f1 'x
                 (fdC 'f2 'y
                      (plusC (idC 'x) (idC 'y))))
            (numC 4))
      (numC 5))
```

它将抛出异常告诉我们没找到标识符 `x` 绑定的值！

但是，它不是应该通过函数 `f1` 的调用被绑定吗？清晰起见，我们切换为（假定的）Racket语法：

```
((define (f1 x)
   ((define (f2 y)
      (+ x y))
    4))
 5)
```

在调用外层函数时，x应该被替换成5，结果是：

```
((define (f2 y)
   (+ 5 y))
 4)
```

继续调用、替换得到 `(+ 5 4)` 也就是 `9` ，并没有出错。

换一种说法，我们肯定是某个地方做错了以至于没有捕捉到函数调用时的参数替换。【注释】函数值需要记住调用过程中执行的替换操作。由于我们使用环境来表示这种替换，因此函数值需要包含记录了该替换的环境。这样得到的数据结构称为闭包（**closure**）：

> 另一方面，如果我们使用替换模型， `x` 会被替换成 `(numV 4)` ，函数体就变成 `(plusC (numV 5) (idC 'y))` ，而它并没有合适的类型。换一种说法，替换模型假设返回值的类型是合法语法。其实尊崇该假设也能学习很多高级编程概念，只是我们不打算往这个方向继续讨论。

注意一下，在解释器的 `appC` 子句中用到了 `funV-arg` 和 `funV-body` ，但是没用到 `funV-name` 。想一下我们之前为什么需要名字这种东西？因为需要通过名字找到函数。但是这里我们通过解释器找到函数，函数名只是作为描述性的存在罢了。换一种说法，函数并不需要名字，就跟常数一样：我们每次使用3的时候并不需要给它命名，那么对于函数为什么要呢？函数本质上是匿名的，我们也应该将其定义和命名分开来。

（但是你可能会说，这种论点只在函数直接定义并使用的情况才成立。如果我们想在某个地方定义，然后在其它地方使用它，我们不还是需要名字的么？是的，正是，后面的匿名之上的语法糖中会说到这个主题）

7.3 实现闭包

首先将函数值类型改为闭包结构体，而不仅仅是函数本体：

```
<answer-type> ::=  ;返回值类型

    (define-type Value
      [numV (n : number)]
      [closV (arg : symbol) (body : ExprC) (env : Env)])
```

同时，我们可以修改函数类型，去除没用的函数名部分。由于历史原因，该构造被称为**lambda**：

```
<fun-type> ::=  ;函数类型

    [lamC (arg : symbol) (body : ExprC)]
```

现在，当解释器遇到函数时，需要记录下到目前为止进行过的所有替换：【注释】

```
<fun-case> ::=  ;函数子句

    [lamC (a b) (closV a b env)]
```

> “Save the environment! Create a closure today!” —Cormac Flanagan

然后在调用函数时，需要使用这个保存下来的环境，而不是空白环境。

```
<app-case> ::=  ;调用子句

    [appC (f a) (local ([define f-value (interp f env)])
                  (interp (closV-body f-value)
                          (extend-env (bind (closV-arg f-value)
                                            (interp a env))
                                      (closV-env f-value))))]
```

事实上这段代码还可以有另一个选择：使用函数调用处的环境：

```
[appC (f a) (local ([define f-value (interp f env)])
              (interp (closV-body f-value)
                      (extend-env (bind (closV-arg f-value)
                                        (interp a env))
                                  env)))]
```

思考题

> 如果我们使用动态的环境（即函数调用处的环境），会导致什么？

回过头来看，现在可以理解为何我们在解释函数体时使用空白环境了。如果函数是定义在程序顶层的，那么它就没有“包含”任何的标识符。因此我们之前的函数实现是现在这种的特殊情况。

7.4 再次聊聊替换

我们已经看到，通过替换这种非常符合直觉的方式可以帮助理解如何实现 `lambda` 函数。然而，对于替换本身我们需要小心一些陷阱！考虑下面这个函数（这里使用Racket语法）：

```
(lambda (f)
  (lambda (x)
    (f 10)))
```

假设 `f` 被替换为lambda表达式 `(lambda (y) (+ x y))` 。注意这里有个自由变量 `x` ，所以如果它被求值，我们应该会得到未绑定变量错误。但是使用替换模型，我们将得到：

```
(lambda (x)
  ((lambda (y) (+ x y)) 10))
```

自由变量消失了！

这是由于我们的替换操作实现的太过简单。为了避免这种异常情况（这也是动态绑定的一种形式），我们需要实现非捕获型的替换（**capture-free substitution**）。大致来说它是这样工作的：我们总是将绑定标识符重命名为从未用过的（新鲜的，**fresh**）名字。比如说，我们给每个标识符加个数字后缀来保证不会出现重名：

```
(lambda (f1)
  (lambda (x1)
    (f1 10)))
```

（请注意，我们把f的绑定和被绑定出现都替换成了f1。）接下来对被替换的表达式也进行同样的重命名：

```
(lambda (y1) (+ x y1))
```

于是替换 `f1` 得到：【注释】

```
(lambda (x1)
  ((lambda (y1) (+ x y1)) 10))
```

> 这里为什么不对作为 `x` 进行重命名呢？因为它可能是引用全局的定义，要么我们也对全局定义进行同样的重命名。这就是所谓的一致性重命名原则。对这个例子来说，这没啥区别。

现在， `x` 仍然是自由变量！这才是正确的替换方式。

等一等。怎么使用环境模型解释器处理这个例子，后果是啥？

思考题

> 试一下。

试了你就知道，一切正确：程序报告有未绑定变量。环境模型实际上实现了非捕获型替换。

练习

> 使用环境是怎么避免替换中的捕获问题的？

7.5 匿名之上的语法糖

让我们回过头考虑函数命名问题，对于实际编程来说它有明显的价值。注意我们现在已经有命名东西的方法：通过函数的调用，参数的值和参数名构成了局部绑定关系。在函数体中，我们只需要用形参就可以引用实参了。

所以说，我们可以用函数来给一系列函数定义命名。例如，考虑Racket代码：

```
(define (double x) (+ x x))
(double 10)
```

等价于：

```
(define double (lambda (x) (+ x x)))
(double 10)
```

一种方法是直接内联（inline）double的定义。不过为了保留命名过程，我们让其等价于：

```
((lambda (double)
   (double 10))
 (lambda (x) (+ x x)))
```

这种模式——我们暂且称为“left-left-lambda”——实际上是种局部命名方式。它非常有用，以至于Racket为它提供了专门的语法：

```
(let ([double (lambda (x) (+ x x))])
  (double 10))
```

`let` 可以通过定义成上面那种语法糖来实现。

下面是个稍微复杂点的例子：

```
(define (double x) (+ x x))
(define (quadruple x) (double (double x)))
(quadruple 10)
```

这可以被改写成：

```
(let ([double (lambda (x) (+ x x))])
  (let ([quadruple (lambda (x) (double (double x)))])
    (quadruple 10)))
```

一切正常。改变一下顺序就不行了：

```
(let ([quadruple (lambda (x) (double (double x)))])
  (let ([double (lambda (x) (+ x x))])
    (quadruple 10)))
```

这是由于 `quadruple` 中“看不见” `double` 。这里我们也能看到全局绑定和局部绑定的区别：位于顶层的全局绑定有“无限的作用域”。这是其强大的地方也是问题的来源。

下面还有个更为微妙的问题，和递归有关。考虑如下的简单无限循环程序：

```
(define (loop-forever x) (loop-forever x))
(loop-forever 10)
```

转换成 `let` ：

```
(let ([loop-forever (lambda (x) (loop-forever x))])
  (loop-forever 10))
```

看上去好像没毛病，是吧？重写成 `lambda` 的形式：

```
((lambda (loop-forever)
   (loop-forever 10))
 (lambda (x) (loop-forever x)))
```

显然，最后一行中的 `loop-forever` 没有被绑定！

对于全局绑定这个问题就不存在。该怎么理解呢？这需要我们理解递归的含义。很快我们将揭开这层神秘的面纱。

8 可变结构体和变量

游戏又来了

下列各表达式哪些意义相同的？

- `f = 3`
- `o.f = 3`
- `f = 3`

假设都是使用Java书写。第一个和第三个的意义可能一样，也可能和第二个相同：完全取决于 `f` 是局域标识符（比如参数）还是对象的字段（如，作为 `this.f = 3` 的简写）。

不管是哪种情况，求值器都将永久改变绑定到 `f` 的值。这对其他观察者而言影响很大。到目前为止，我们实现的计算过程对于相同的输入总是给出相同的输出。现在计算的答案还取决于它在何时进行：在 `f` 的值改变前还是后。时间的引入对于代码的推理有深远影响。

此外，上述简单的语法包含了两种不同的改变：改变字段的值（ `o.f = 3` 或者 `this.f = 3` ）和改变标识符的值（ `f = 3` ，其中 `f` 在方法内部被绑定而不是由对象绑定）有着非常大的区别。我们会依次讨论它们。首先探讨字段，再在变量那一节中探讨标识符。

8.1 可变结构体

8.1.1 可变结构体的简化模型

很快我们会带大家认识到，对象其实就是一般化的结构体。对象中的字段可认为是结构体中字段的一般化的结果。要理解赋值，理解可变对象大致足够了（并不完全足够）。为了简单起见，我们甚至不需要结构体具有多个字段：一个字段就足够了。我们称该结构为**box**。在Racket中，box仅支持三种运算：

```
box : ('a -> (boxof 'a))
unbox : ((boxof 'a) -> 'a)
set-box! : ((boxof 'a) 'a -> void)
```

`box` 接受一个值，将其包裹在可变容器中。 `unbox` 取出容器中的当前值。 `set-box!` 改变容器中的值，对于静态类型的语言来说，新值需要和旧值保持类型一致。如果对应到Java中的话，box大致等价于带类型参数的Java容器类，只有一个字段，外加getter和getter: `box` 对应构造器， `unbox` 对应 `getter` ， `set-box!` 对应 `setter` （由于只有一个字段，所以字段名也无所谓了）：

```
class Box<T> {
    private T the_value;
    Box(T v) {
        this.the_value = v;
    }
    T get() {
        return the_value;
    }
    void set(T v) {
        the_value = v;
    }
}
```

由于赋值操作经常成组进行（例如，从银行账户中取出一些钱存放到另一个账户中），支持赋值操作的序列将非常有用。在Racket中，你可以使用 `begin` 表示操作的序列；它将依次计算序列中的每个表达式然后返回最后一个的求值结果。

练习

> 尝试使用 `let` 对 `begin` 去语法糖（还可以进一步去语法糖到 `lambda` ）。

尽管可以将 `begin` 当作语法糖（从核心语言中）去除，但是它对理解赋值的内部原理非常有用。因此我们还是决定直接在核心语言中支持简单的 `begin` ，该 `begin` 形式只允许两个子项。

> 这也说明，去语法糖没有绝对的规范。我们选择在核心语言中加上这个构造，而它并不是必须的。如果我们的目的是尽可能减小解释器的体积——即使增大输入程序的体积也在所不惜——那么就不应该这么做。不过我们在本书中的目的是学习（适合教育目的的）解释器，那么选择大一点的语言更加有指导性。

8.1.2 脚手架

首先，扩展语言的核心数据类型：

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [appC (fun : ExprC) (arg : ExprC)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [lamC (arg : symbol) (body : ExprC)]
  [boxC (arg : ExprC)]
  [unboxC (arg : ExprC)]
  [setboxC (b : ExprC) (v : ExprC)]
  [seqC (b1 : ExprC) (b2 : ExprC)])  ;序列
```

注意 `setboxC` 表达式中，两个操作对象均为表达式。值（v）为表达式很自然，没什么奇怪的；但是 `box` 参数（b）为表达式的话乍一看还挺奇怪的。它意味着我们可以写出对应于如下Racket代码的程序：

```
(let ([b0 (box 0)]
      [b1 (box 1)])
  (let ([l (list b0 b1)])
    (begin
      (set-box! (first l) 1)
      (set-box! (second l) 2)
      l)))
```

其计算结果为 `box` 的链表，第一个box包含的值为 `1` ，第二个包含的值为 `2` 。【注释】观察程序中第一个 `set-box!` 指令，其第一个参数为 `(first l)` ，也就是说，是计算结果为 `box` 的表达式，而不是字面的 `box` 也不是标识符。和Java中下列代码类似（放松类型要求）：

```
public static void main (String[] args) {
    Box<Integer> b0 = new Box<Integer>(0);
    Box<Integer> b1 = new Box<Integer>(1);

    ArrayList<Box<Integer>> l = new ArrayList<Box<Integer>>();
    l.add(b0);
    l.add(b1);

    l.get(0).set(1);
    l.get(1).set(2);
}
```

> 输出可能是 `'(#&1 #&2)` 。 `#&` 是Racket中box类型的语法缩写形式。

注意到其中 `l.get(0)` 为复合表达式，它得到一个 `box` 对象，然后调用其 `set` 方法。

为方便起见，我们假设已经实现了下列去语法糖操作：

1. `let`
2. 必要的话，多于两个子项的序列（可以去语法糖为嵌套的序列）

有时我们还会直接使用Racket语法写程序，一方面是为了简洁（我们的核心语言将变得大而笨重），一方面方便你可以直接在Racket中运行相关代码观察结果。也就是说，我们会使用Racket（大部分主流语言中可变对象和结构体行为都与之类似）作为我们实现的参照。

8.1.3 与闭包的交互

考虑如下的简单计数器：

```
(define new-loc
  (let ([n (box 0)])
    (lambda ()
      (begin
        (set-box! n (add1 (unbox n)))
        (unbox n)))))
```

每次调用，它都会返回下一个自然数：

```
> (new-loc)
- number
1
> (new-loc)
- number
2
```

为什么会这样呢？这是因为其中的 `box` 只被创建了一次，它被绑定到了 `n` ，然后该绑定被放进闭包。所有后续的赋值操作改变的都是同一个 **`box`** 。如果交换两行代码，结果就完全不同了：

```
(define new-loc-broken
  (lambda ()
    (let ([n (box 0)])
      (begin
        (set-box! n (add1 (unbox n)))
        (unbox n)))))
```

运行看看：

```
> (new-loc-broken)
- number
1
> (new-loc-broken)
- number
1
```

这种情况下，每次调用函数都会创建新的 `box` ，所以每次的计算结果都是一样的（尽管程序内部也变动了 `box` 的值）。我们对于 `box` 的实现也应该正确重现这种区别。

上面的例子给了我们一点关于实现上的提醒。显然， `new-loc` 的闭包中每次引用的必须是同一个 `box` 。然而我们还需要做些工作来确保获得的 `box` 中的值每次都是不同的！请仔细体会：它从词法上来看必须是相同的，但是动态的值却是不同的。这个区分将是我们实现的核心。

8.1.4 理解box的解释

首先重现一下当前的解释器：

```
<interp-take-1> ::=  ;解释器，第一次尝试

    (define (interp [expr : ExprC] [env : Env]) : Value
      (type-case ExprC expr
        [numC (n) (numV n)]
        [idC (n) (lookup n env)]
        [appC (f a) (local ([define f-value (interp f env)])
                      (interp (closV-body f-value)
                              (extend-env (bind (closV-arg f-value)
                                                (interp a env))
                                          (closV-env f-value))))]
        [plusC (l r) (num+ (interp l env) (interp r env))]
        [multC (l r) (num* (interp l env) (interp r env))]
        [lamC (a b) (closV a b env)]
        <boxC-case>    ;box子句
        <unboxC-case>  ;unbox子句
        <setboxC-case> ;setbox子句
        <seqC-case>))  ;序列子句
```

由于引入了新类型的值——box，我们需要更新返回值的数据类型：

```
<value-take-1> ::=  ;值，第一次尝试

    (define-type Value
      [numV (n : number)]  ;数
      [closV (arg : symbol) (body : ExprC) (env : Env)]  ;闭包
      [boxV (v : Value)])
```

先实现两种简单的情形。对于 `box` 表达式，直接求值并使用 `boxV` 包裹后返回：

```
<boxC-case-take-1> ::=  ;box子句，第一次尝试

    [boxC (a) (boxV (interp a env))]
```

同样，从 `box` 中提取值也很简单：

```
<unboxC-case-take-1> ::=  ;unbox子句，第一次尝试

    [unboxC (a) (boxV-v (interp a env))]
```

到这里你应该已经写过一组测试，来保证新加代码行为同预期一样。

当然，现在还没有做到难的部分。可以预见，所有有意思的行为都在对 `setboxC` 的处理上。然而，我们却要先考察 `seqC`（你会看到我们为什么把它加到核心语言中）。

先试试二目序列最自然的实现方式：

```
<seqC-case-take-1> ::=  ;序列子句，第一次尝试

    [seqC (b1 b2) (let ([v (interp b1 env)])
                    (interp b2 env))]
```

即先计算第一个子项，然后计算第二个子项并返回其计算结果。

你应当迅速察觉到一些问题，我们计算了第一个子项并把它的值绑定到了`v`，但是后面的计算过程中没有用它。这倒没关系：正常来说，第一个子项中包含了某种赋值操作，其返回值没啥用（确实，注意`set-box!`返回void值）。那么我们可以实现如下：

```
<seqC-case-take-2> ::=  ;序列子句，第二次尝试

    [seqC (b1 b2) (begin
                    (interp b1 env)
                    (interp b2 env))]
```

这种实现并不令人满意，它直接使用了Racket中的序列操作（无助于我们理解），更严重的问题是，它不可能是正确的！因为，我们必须要把赋值操作的结果存储起来。但是，我们的解释器只能求出表达式的值，任何在`(interp b1 env)`中进行的赋值操作都将丢失。显然这不是我们想要的。

8.1.5 环境能帮我们解决问题吗？

下面这个例子能给我们一点启示：

```
(let ([b (box 0)])
  (begin (begin (set-box! b (+ 1 (unbox b)))
                (set-box! b (+ 1 (unbox b))))
         (unbox b)))
```

在Racket中，它求值得`2`。

练习

> 使用`ExprC`表示该表达式。

考虑内层的`begin`的求值过程。它的两个子项（`(set-box! ...)`的`ExprC`表示）完全相同。然而幕后肯定有什么东西悄悄改变了，因为`box`中的值会从0变成2！上面的例子修改一下我们能“看”得更清楚：

```
(let ([b (box 0)])
  (+ (begin (set-box! b (+ 1 (unbox b)))
            (unbox b))
     (begin (set-box! b (+ 1 (unbox b)))
            (unbox b))))
```

这下求值得到3。这里，当处理到加法时，需要对两个操作数调用两次`interp`，传给它们的表达式是完全相同的。然而，第一个调用的行为显然会被第二个调用感知到。我们需要解开背后的魔法。

如果给解释器输入了两个一模一样的表达式，它返回的结果怎么会不一样呢？最简单的解释，解释器的另一个参数，即环境，发生了某些变化。我们现有的解释器在处理加法时，对俩个操作数调用`interp`时用的环境是一样的；在处理序列时，对两个子项调用`interp`时用

的环境也是一样的。所以现有的解释器，是不可能产生我们想要的结果的——相同的输入总是会得出相同的输出。

通过上述例子我们得到的一些启示：

1. 多次调用解释器，并且我们认为其返回值可能不同的情况下，我们需要确保传递给解释器的参数也不同
2. 解释器需要返回一种记录，其中保存了求值过程中进行过的赋值

由于输入的表达式不可能改变，所以第一条指引我们使用环境来反映不同调用之间的不同。结合第二点我们很自然的想到让解释器返回环境，然后可以将它传递给下一个调用。于是，大致来说解释器的类型可能就变成：

```
; interp : ExprC * Env -> Value * Env
```

即，解释器接收表达式和环境作为参数；在该环境中求值，同时求值过程中更新环境；计算完成后（和以前一样）返回求值结果，同时还返回更新后的环境。新的环境被传入解释器的下一次调用中。`setboxC` 的处理过程中应该会影响到环境，以反应它所执行的赋值操作。

在着手实现之前，我们应先考虑这种改变的后果。环境已经负担了重任：保存被延迟的替换操作的所需的信息。它已经有非常明确的语义——由替换给定——我们应该注意，不要影响这层语义。它和替换之间的这种关系使得它成为了词法作用域的信息仓库。如果我们扩展环境的功能，使得加法的一个参数分支中的绑定通过它可以传递到另一个参数分支中，例如，考虑下面的程序：

```
(+ (let ([b (box 0)])
     1)
   b)
```

显然该程序将报错：加法的第二个参数 `b` 是未绑定的（`b` 的作用域终止于 `let` 表达式的终结——如果上面的代码对你来说不够清晰，用函数把 `let` 语法糖去除）。但是，如果扩展了环境的功能，解释完第一个参数后产生的环境中显然包含了 `b` 的绑定信息。

练习

> 尝试使用已有的解释器的逻辑运行这段代码，以确保真正理解上面表达的意思。

当然你可能考虑其它实现方式，不过它们一般来说都会导致类似的失败。比如你可能会想，由于问题出在多余的绑定上，我们可以将返回的环境中多余的绑定直接移除。听上去不错，但是你还记得我们还需要实现闭包吗？

练习

> 考虑如下程序的`ExpC`表示：
>
> ```
> (let ([a (box 1)])
>   (let ([f (lambda (x) (+ x (unbox a)))])
>     (begin
>       (set-box! a 2)
>       (f 10))))
> ```
>
> 看看这个方案有啥问题。

要认识到，前面提到的两个启示中的约束都是有效的，但是解决方案并不在上面提出的这些尝试中。再仔细想想，那两个启示中所提出的约束都没要通过环境去实现。而且环境显然也没法负起这个职责。

8.1.6 引入贮存

通过上一节的讨论，我们意识到需要额外的仓库来记录表达式的解释过程。仓库之一是环境，还是执行本来赋予它的职责，维护词法作用域。但是环境不能直接将标识符映射到值，因为现在值是可能会变的。也即，我们需要额外的东西用于维护可变`box`的动态状态，这个额外的东西被称之为贮存（store）。

和环境一样，贮存也是映射结构。它的值域可以是任意的名字的集合，不过自然的想法是将其想作用于表示内存地址的数。这是因为，在语义上来说，存储就对应于（抽象的）计算机的物理内存，而传统上内存地址一般采用数进行寻址。因此环境是将名字映射到地址，然后贮存将地址映射到具体的值。

```
(define-type-alias Location number)  ;地址

(define-type Binding  ;绑定
  [bind (name : symbol) (val : Location)])

(define-type-alias Env (listof Binding))  ;环境
(define mt-env empty)
(define extend-env cons)

(define-type Storage  ;贮存物
  [cell (location : Location) (val : Value)])

(define-type-alias Store (listof Storage))  ;贮存
(define mt-store empty)  ;空贮存
(define override-store cons)  ;覆盖贮存
```

我们还需要提供函数用于在贮存中查询值，就跟之前的环境一样（现在环境中查询的结果是地址了）。

```
(define (lookup [for : symbol] [env : Env]) : Location
  ...)
(define (fetch [loc : Location] [sto : Store]) : Value
  ...)
```

有了这些，就能完成解释器返回值的正确表示了：

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)]
  [boxV (l : Location)])
```

练习

> 完成查询函数`lookup`和获取函数`fetch`的函数体部分。

8.1.7 解释器之解释`box`

现在有了贮存，环境可以返回之、可以更新之从而反映求值过程中的赋值，而且赋值本身不需要修改环境中的内容。由于函数只能返回一个值，我们考虑定义一个数据结构用于存放解释器的返回值：

```
(define-type Result  ;结果
  [v*s (v : Value) (s : Store)])
```

于是，解释器的类型变成了这样：

```
<interp-mut-struct> ::=  ;解释器，可变结构体

    (define (interp [expr : ExprC] [env : Env] [sto : Store]) : Result
      <ms-numC-case>
      <ms-idC-case>
      <ms-appC-case>
      <ms-plusC/multC-case>
      <ms-lamC-case>
      <ms-boxC-case>
      <ms-unboxC-case>
      <ms-setboxC-case>
      <ms-seqC-case>)
```

数的解释依然是最简单的。记住我们需要返回贮存，该贮存反映求值输入表达式过程中所发生的全部赋值。由于数是常量，求值过程不会有赋值发生，所以，直接返回传入的贮存即可：

```
<ms-numC-case> ::=

    [numC (n) (v*s (numV n) sto)]
```

创建闭包也是一样；注意是闭包的创建而不是调用：

```
<ms-lamC-case> ::=

    [lamC (a b) (v*s (closV a b env) sto)]
```

标识符的处理很直接。当然如果你的实现过于简单，类型系统会告诉你错在哪里：为了获取返回值，你即要查询环境也要查询贮存：

```
<ms-idC-case> ::=

    [idC (n) (v*s (fetch (lookup n env) sto) sto)]
```

注意到 `lookup` 和 `fetch` 组合在一起完成之前由 `lookup` 完成的工作。

接下来的事情才有意思呢。

考虑序列的处理。显然，我们需要解释两个子项：

```
(interp b1 env sto)
(interp b2 env sto)
```

等一下。我们的目的是，当对第二个子项求值时使用第一个子项返回的贮存——否则这么多改变就毫无意义了。因此我们必须先对第一个子项求值，获取其返回的贮存，用它对第二个贮存的求值：

```
<ms-seqC-case> ::=

    [seqC (b1 b2) (type-case Result (interp b1 env sto)
                    [v*s (v-b1 s-b1)
                         (interp b2 env s-b1)])]
```

先调用 `(interp b1 env sto)`，其返回的值和贮存被分别命名为 `v-b1` 和 `s-b1`；接下来使用新的贮存对第二个子项求值：`(interp b2 env s-b1)`。它的返回值该子项的值和贮存，正好是我们需要的东西。代码也可以反映出，第一个子项的唯一效果就是其返回的贮存：虽然我们绑定了 `v-b1` 但后文并没有用到它。

思考题

> 你可以多花点时间玩味一下这段代码。后面将经常用到该种模式的代码。

下面来处理双目算术运算。它们和序列的求值类似，也含有两个子项要处理，但是这里我们还需要用到两个子项各自的值。和以前一样，我们只给出 `plusC`，`multC` 的代码基本上相同：

```
<ms-plusC/multC-case> ::=

    [plusC (l r) (type-case Result (interp l env sto)
                   [v*s (v-l s-l)
                        (type-case Result (interp r env s-l)
                          [v*s (v-r s-r)
                               (v*s (num+ v-l v-r) s-r)])])]
```

同样的模式这里用了两层，以便我们分别取得两个返回值，然后将其传给 `num+`。

这里可以看到环境和贮存的重要区别。当对子项求值时，根据语言的作用域规则，通常所有子项都使用相同的环境。环境的传递遵从递归向下的模式。与之相对，贮存是线式传递的：所有的分支并不使用同一个贮存，前一个分支产生的贮存后一个分支使用，最后一个分支的

贮存就是总的返回贮存。这种风格被称作贮存传递模式（**store-passing style**）。

现在谜题彻底揭晓，贮存传递模式就是我们的秘密神器：它在保障环境依旧正确处理词法作用域的同时，给了我们能够记录赋值操作的方法。直觉告诉我们，环境肯定参与这个过程，同一个表达式可以返回不同的值，现在我们可以看清这是怎么做到的了：不是直接修改环境实现，而是环境间接的引用了贮存，而贮存会更新。下面我们需要看看贮存是如何“更新”自己的。

首先考虑将值放到 `box` 中。我们得分配一块地方让贮存放东西。 `box` 的值会记住该地址，用于之后 `box` 的赋值操作。

```
<ms-boxC-case> ::=

    [boxC (a) (type-case Result (interp a env sto)
                [v*s (v-a s-a)
                     (let ([where (new-loc)])
                       (v*s (boxV where)
                            (override-store (cell where v-a)
                                            s-a)))])]
```

思考题

> 注意了注意了，上面的代码依赖于 `new-loc` ，而 `new-loc` 的实现中又用到了 `box` 。这就很尴尬了。你能不能修改解释器，使其不再依赖于类似于 `new-loc` 这种本身需要赋值的东西？

要消除 `new-loc` 这种类型的东西，最简单的方式是再给解释器添加参数和返回值，用于表示当前使用过的最大地址。每次分配贮存地址的操作都会返回递增过的地址，而其它操作则直接返回原最大地址。换一种说法，我们又用了一次贮存传递模式。这样去实现的话解释器会显得太笨拙，以至于掩盖更重要的内容：用贮存传递模式实现贮存。这也就是为啥这里我们没这么做的原因。但是，我们必须明白这么做是可行的：不依赖于 `box` 而在我们的语言中实现 `box` 。

由于 `box` 记录内存地址，获取 `box` 中的值比较简单：

```
<ms-unboxC-case> ::=

    [unboxC (a) (type-case Result (interp a env sto)
                  [v*s (v-a s-a)
                       (v*s (fetch (boxV-l v-a) s-a) s-a)])]
```

用到了同样的模式，具体来说我们调用 `fetch` 来获取该地址中的实际值。注意这里的代码没有判断 `a` 的求值结果是否是 `boxV` ，而是依赖于宿主语言Racket在不是时抛出异常；如果是别的宿主语言，不进行该类型判断就可能很危险了（比如C语言，相当于允许访问任意内存）。

下面考虑怎么更新 `box` 中的值。首先还是要求值得到 `box` 和要放入的新值。 `box` 的值将为 `boxV` 类型，其中含有地址。

原则上，我们是要“改变”，或者说覆盖贮存中对应地址上的值。有两种方式可以实现这点：

1. 遍历贮存，找到对应地址的绑定，然后替换该地址上绑定的值，贮存中的其它绑定保持不变。
2. 懒一点的做法，直接给贮存新增绑定，而查询贮存时只查找最新的绑定即可（就跟环境中 `lookup` 函数的实现一样，没有理由 `fetch` 不这么干）。

两种选择都不会影响到下面的代码：

```
<ms-setboxC-case> ::=

    [setboxC (b v) (type-case Result (interp b env sto)
                     [v*s (v-b s-b)
                          (type-case Result (interp v env s-b)
                            [v*s (v-v s-v)
                                 (v*s v-v
                                      (override-store (cell (boxV-l v-b)
                                                            v-v)
                                                      s-v))])])]
```

当然，由于前面 `override-store` 的实现就是 `cons` 而已，我们实际上使用的是比较偷懒的方式（而且是有风险的选择，因为它还取决于 `fetch` 的实现）。

练习

> 实现另一种方式的贮存更新，更新原有的绑定关系，避免贮存中出现相同地址的多个绑定。

练习

> 在更新步骤中，当我们查找贮存中的地址时，是否可能发生找不到某个地址的情况？如果可能，请编写程序演示这种情况。如果不能，请指出解释器的哪个不变量避免了这种情况的发生。

好了，现在我们只差函数调用的情况了！函数调用的整体流程我们已经很熟悉了：求值函数部分，求值参数部分，扩展闭包的环境，然后再其中求值闭包的函数体部……但是贮存是如何参与这一切的呢？

```
<ms-appC-case> ::=

    [appC (f a)
          (type-case Result (interp f env sto)
            [v*s (v-f s-f)
                 (type-case Result (interp a env s-f)
                   [v*s (v-a s-a)
                        <ms-appC-case-main>])])]  ;调用子句主体
```

从如何扩展闭包的环境入手好了。新增绑定的名字显然应该是函数的形参；但是它应该被绑定到什么地址呢？为了避免使用已有地址将招致的困惑（我们后面将详细介绍会招致何种困惑！），先使用新分配的地址吧。将该地址绑定到环境中，然后将求得的参数值存放在贮存的该地址上：

```
<ms-appC-case-main> ::=  ;调用子句主体

    (let ([where (new-loc)])
      (interp (closV-body v-f)
              (extend-env (bind (closV-arg v-f)
                                where)
                          (closV-env v-f))
              (override-store (cell where v-a) s-a)))
```

我们也没说要把函数参数实现为可变的，所以其实也没必要这么实现函数调用。事实上使用跟以前一样的策略没有任何问题。观察一下，在上面这种实现中，这个地址中的值也不会被修改：只有 `setboxC` 能够改变现有地址的内容（严格来讲 `override-store` 只是对贮存的初始化），而且只能改变 `boxV` 中的数据，但是这里并没有创建 `box` 。我们这么实现是出于统一的考虑，并且这么做还可以减少需要处理的子句。

练习

> 将贮存地址限制为只能被 `box` 使用是很好的练习。有哪些代码需要改动？

8.1.8 回顾思考

尽管完成了解释器的实现，仍然还有不少微妙的问题和一些洞察值得拿出来讨论一下。

1. 我们的解释器实现中隐藏了一个巧妙但重要的设计抉择：求值的顺序。例如，为什么我们不按如下方式实现加法？

    > ```
    > [plusC (l r) (type-case Result (interp r env sto)
    >                [v*s (v-r s-r)
    >                     (type-case Result (interp l env s-l)
    >                       [v*s (v-l s-l)
    >                            (v*s (num+ v-l v-r) s-l)])])]
    > ```

    事实上这样做也是自洽的。类似地，贮存传递模式中蕴含了先计算函数部分再计算参数部分这种抉择。注意到：

    - 以前，这种抉择直接代理给了宿主语言的实现，现在，贮存传递迫使我们把计算过程顺序化，因此该抉择是由我们自己作出的（不管是有意还是无意）。
    - 更为重要的是，现在这是语义上的抉择了。在没有赋值之前，加法一个分支上的计算不会影响另一个分支上的计算结果。【注释】而现在，分支上可能会执行赋值操作从而因此影响到另一分支，因此要使该语言的程序员能预测自己程序的行为，我们必须选择某种求值顺序！明确地写出贮存传递解释器也表明了这一点。
2. 观察函数调用的规则，可以发现，我们往下传递的是动态的贮存，即，先后经过了计算函数和计算参数的那个贮存。这种行为跟我们对于环境的要求正好相反。这是个关键的区别。贮存从其效果上来说，是“动态作用域的（dynamically scoped）”，这是由于它是用于反映计算的历史，而不是用来反映词法上的东西。由于我们已经使用了名词“作用域（scope）”来表示标识符的绑定，这时再用“动态作用域的”来描述贮存可能会造成困惑。于是我们引入新名词持久的（**persistent**）来描述贮存。

一些语言中这两个概念混淆不清。例如在C语言中，绑定到局域标识符上的值（默认）在堆栈上分配。然而，堆栈对应于这里的环境，因此它们将随着函数调用的结束而消失。如果函数返回值中引用了这些值，那么这个引用将会指向某个未使用的地址，或者被用作他用的地址：C语言中很大一部分错误来源于此。问题的关键是，值本身不会消失；消失指向它们的、具有词法作用域的标识符。

3. 我们已经讨论过两种实现覆写贮存的策略：简单的扩展之（将依赖于`fetch`的实现，需要它总是取出最新的绑定）；或者采用"搜索替换"的方式。后面这种策略有个好处，不会存储那些无用的、永远不可能访问得到的数据。

   然而这么做还是会浪费内存。随着程序的运行，我们会永久失去访问某些`box`的能力：例如，某个`box`仅被绑定到一个标识符上，程序走出该标识符的作用域后（将再也不能访问到该`box`）。这些不能被访问到的位置被称为垃圾（garbage）。从概念上来讲，垃圾地址是那些清除之后对程序求值结果没有任何影响的地址。有很多用于辨别并回收垃圾的策略，通常被称作垃圾回收（garbage collection）。

4. 要注意，计算表达式的时候，总是要让后面的计算依赖之前返回的贮存以维护正确的执行历史。比如，考虑下面这种`unboxC`的实现：

   > ```
   > [unboxC (a) (type-case Result (interp a env sto)
   >               [v*s (v-a s-a)
   >                    (v*s (fetch (boxV-l v-a) sto) s-a)])]
   > ```
   >
   > 注意到区别没有？我们没有从`s-a`而是从`sto`中获取值。但`sto`反映的是`unboxC`未求值之前的赋值历史，而没有包含它求值过程中的赋值历史。`unboxC`表达式求值过程中贮存可能发生改变吗？当然了！
   >
   > ```
   > (let ([b (box 0)])
   >   (unbox (begin (set-box! b 1) b)))
   > ```
   >
   > 如果按照上面这种错误的实现，它将得到0而不是正确的值1。

5. 下面是另一个类似的错误：

> ```
> [unboxC (a) (type-case Result (interp a env sto)
>                [v*s (v-a s-a)
>                     (v*s (fetch (boxV-l v-a) s-a) sto)])]
> ```
>
> 什么例子程序可以展示其错误呢？注意到，它返回的是原始的贮存，未经`unboxC`求值过程修改。所以我们需要在后续代码中访问贮存：
>
> ```
> (let ([b (box 0)])
>   (+ (unbox (begin (set-box! b 1)
>                    b))
>      (unbox b)))
> ```
>
> 它本应求值得2，但是由于返回的贮存中b的值一直绑定为0，导致结果为1。

如果把前述二点中的错误结合起来——解释器子句中最后一行两次都使用`sto`而不是`s-a`——该表达式的结果将变成0.

练习

> 将解释器中所有贮存，逐一替换为更新前的贮存；对每一个这样的修改，给出能够显示其错误的测试案例；请确保你最后得到覆盖所有情况的测试案例集。

6. 观察前述对"旧"贮存的使用，它允许我们进行时间回溯：赋值引入了时间的概念；使用原先的贮存则允许我们回到过去，也就是赋值没有发生之前。这听起来一方面蛮有趣另一方面有悖常情；它有合理用途吗？

   有！想象一下，我们不直接改变贮存，而是引入日志的概念，表示贮存中意向中的更新。日志的实现方式类似于贮存，线性传递。（语言中）添加创建新日志的指令；对于查询操作，首先检查日志，仅当日志中找不到某个地址的绑定时，才在实际贮存中查找。还要添加两个新指令：丢弃（discard）某个日志（用于进行时间回溯），以及 提交（commit）操作（将某个日志中的修改全部应用到贮存中）。

   事实上这就是软件事务内存（Software Transactional Memory）的概念。（每条线程都只能看到自己的日志和全局的贮存，看不到其他线程的日志，）其他线程在提交日志之前所做的修改对本线程是透明的。这就是说，每个线程看到的世界都是一致的（能看到自己所做的修改，因为它们都在日志中）。如果事务成功完成（提交），那么所有线程都都会看到更新后的全局贮存；如果事务中止（丢弃），被丢弃的日志也带走了其中所有的修改，状态还原（其他线程做提交还是会生效）。

   多线程编程会带来很多难题，软件事务内存提供了一种非常合理的解决办法，如果线程间必须共享可变状态的话。大部分计算机都只有一个全局存储，维护日志成本可能会很高，所有人们花了很大精力优化它们。另一种解决方案是，某些硬件架构开始提供对事务内存的直接支持，这使得日志的创建、维护和提交可以和操作全局存储一样高效，移除了采用该想法的一个重大阻碍。

   练习

> 修改语言，增加日志功能以实现软件事务内存。

练习

> 另一种实现策略是，在环境中将名字映射到box类型的值。这里我们没有这样做是因为：
>
> 1. 这样做的话有种作弊的感觉
> 2. 学不到不使用box实现该特性的方法
> 3. 不一定能扩展到其他赋值操作
> 4. 更重要的是，不能让我们获得这些洞见
>
> 不过理解该策略还是很有用的，而且你在实现自己的语言的时候可能会觉得采用这也是个好主意。因此，试试使用这种策略实现一下我们的解释器。你还需要贮存传递模式吗？为什么？
>
> 唯一的影响是，某个分支可能会报错或者永不终止——当然这都是外部可见的影响，但是它们都是更高层次的影响。如果程序正常返回的话，不管选择哪种求值顺序，返回值还都是一样的。）

8.2 变量

搞定了可变结构体，接下来考虑另一种情况：变量赋值。

8.2.1 术语

首先，关于名词的选择。之前我们一直坚持使用“标识符”，这是因为我们想将“变量”留给将要学习的东西。在Java中，当我们写出（这里假设 `x` 为局域绑定的，比如是某个方法的参数）

```
x = 1;
x = 3;
```

我们是在要求改变 `x` 的值。经过第一次赋值之后， `x` 的值为1；第二次之后为3。因此， `x` 的值会在方法的执行过程中变化。

我们在数学中通常也会使用“变量”这个词表示函数参数。例如，在 **`f(y) = y + 3`** 中，我们称 **`y`** 为“变量”。这里它被称为变量是由于不同的调用之间 `y` 的值也不同；然而，在同一次调用内部，在其作用域内它的值总是一样的。之前的标识符对应于这种意义上的变量。【注释】与之相对的，程序变量在每次调用内部都可以变化，如上面Java代码中的 `x` 。

> 如果某个标识符被绑定到一个 `box` ，那么它将总是被绑定到同一个 `box` 值。会发生改变的是 `box` 的内容，标识符和 `box` 的绑定关系不会变。

从今往后，我们使用变量表示在其作用域内值可以发生变化的标识符，而值不能变化的使用标识符表示。如果情况存疑时，安全一点，我们就称之为“变量”；如果这种区分不太重要时，我们也可能使用其中任意一个。不要被这些名词搞得头大，重要的是理解它们的区别。

8.2.2 语法

大部分语言使用 `=` 或者 `:=` 表示赋值，Racket选择了不同的语法：使用 `set!` 进行变量赋值。这就要求Racket程序员直面我们在本章开头所提到的区别。当然，这里我们绕开语法区别，在我们的核心语言中使用不同的结构分别表示box和变量。

关于变量赋值，首先要认识到的是，尽管它和box赋值（setboxC）一样有两个子项，但是两者的语法是完全不同的。为了理解其中区别，先考虑下面的Java代码：

```
x = 3;
```

在这个语句中，`x` 的位置不能为任意表达式：它必须是标识符本身。这是因为，如果该位置为任意表达式，那么我们就必须对其进行求值，然后得到某个值：例如，如果 `x` 之前绑定到1，那就意味着我们将会产生下面这样的式子：

```
1 = 3;
```

但显然这是没意义的！我们不能给1赋值，事实上1就是所谓的不变量。我们想要的是找到 `x` 在贮存中的位置，然后改变该位置上存的值。

再看个例子。假设局域变量 `o` 被绑定到某个字符串对象 `s`，然后我们写出下面的语句：

```
o = new String("a new string")
```

我们是打算修改 `s` 吗？当然不是。该指令应该保持 `s` 不变，我们只是想改变 `o` 指向的值，使得后面程序中 `o` 被求值时得到的是这个新的字符串对象。

8.2.3 解释器之解释变量

首先修改语法：

```
(define-type ExprC
  [numC (n : number)]
  [varC (s : symbol)]
  [appC (fun : ExprC) (arg : ExprC)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [lamC (arg : symbol) (body : ExprC)]
  [setC (var : symbol) (arg : ExprC)]
  [seqC (b1 : ExprC) (b2 : ExprC)])
```

可以看见我们丢弃了 `box` 相关操作，但是保留了序列，因为赋值和序列操作息息相关。注意我们添加的 `setC` 形式，其第一个子项不是表达式而是变量的名字。同时我们还将 `idC` 改作 `varC` 。

由于去掉了 `box` ， `box` 值也不需要了：

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)])
```

可能和你想的一样，为了支持变量，出于和前面相同的原因，我们仍需要用到贮存传递模式（8.1.7节）。区别在于如何使用它。注意到之前序列的实现不需要变动（它并不依赖于要改变的东西是box还是变量），于是就只剩下变量赋值需要处理了。

首先还是要对新值表达式求值，并获取更新后的贮存：

```
<setC-case> ::=

    [setC (var val) (type-case Result (interp val env sto)
                      [v*s (v-val s-val)
                           <rest-of-setC-case>])]  ;setC子句其余部分
```

接下来呢？前面讨论过了，对于变量部分，我们不应对求其值（这么做只会获取其旧值），而是应该获取它对应的存储地址，然后更新该地址中的内容，最后这步和之前box的处理类似：

```
<rest-of-setC-case> ::=  ;setC子句其余部分

    (let ([where (lookup var env)])
      (v*s v-val
           (override-store (cell where v-val)
                           s-val)))
```

这个新模式才是意义所在。在处理box的过程中，对于 `idC` 的处理是：先从环境中找出标识符的地址，然后直接从贮存中获取其值；两步之后得到值，和（在解释器中）增加贮存之前进行查找获得的是一种东西。而现在，新的模式是：对于变量标识符的处理止步于从环境中获取其存储地址（并不继续获取其值）。这样获得的值按按照传统被称为左值，“（赋值语句）左侧的值”之意。这是“存储地址”花哨的说法，它和贮存中存储的眞实值不同：注意到它并不和 `Value` 中任何类型对应。

这个解释器已经完成了！所有的难点已经在之前实现贮存传递模时（包括处理函数调用时，给新变量分配地址）搞定了。

8.3 设计语言时状态的考虑

尽管大部分语言都包含状态，我们所学习的两种状态之一或者两者都有；但是它们的选入不应该被当做一件微不足道或者理所当然的事。一方面，状态的引入带来了明显的好处：

- 状态提供了某种形式的模块化。拿我们上面实现的解释器为例，如果没有显式的状态操作（而要达到同样效果）：
    - 为了传递贮存，需要将其放入所有函数的参数和返回值中
    - 所有可能会涉及到状态的函数都需要修改，维护信息的传递链 可以将编程语言中的状态理解为在所有函数间隐式流动的的参数和返回值，而无需程序员费力地维护。它使得不同函数可以进行“超距”通信，中间子程序无需知晓这种通信。
- 状态得以让我们构造动态、环形的数据结构，或者至少提供了一种简洁直观的方式做到（第九章会讨论）
- 状态赋予子程序内存，比如前述的new-loc。如果某个子程序没法自己记住事情，那么其调用者就必须帮它完成，本质上就是做类似于传递贮存的事情。这么做不仅不方便，还给调用者恶意修改内存的机会（比如说，子程序的调用者可以故意送回旧的贮存，从而获取已经交给其他调用方的引用，通过这种方式发起正确性或安全攻击）。

另一方面，状态也给程序员和处理程序的程序（如编译器）带来不少麻烦。其中一个是“别名（aliasing）”，以后我们会讨论到。另一个是“引用透明（referential transparency）”，也是希望以后我们能讨论到。最后，上面我们说过状态提供了某种形式的模块化。然而，换个角度看，两个子程序之间通过秘密渠道进行了通信，而它们的中间人无法获知也无法监控这种通信。某些情况下（特别是安全系统和分布式系统中），这种秘密渠道非常危险，也不受欢迎。

没有完美的方案，所以一种明智的选择是，提供赋值操作，同时又对其区别对待。例如，Standard ML中没有变量，因为它被认为不是必要的。但是该语言包含了等价于 `box` 的东西（叫做ref(引用)）。你可以很容易的用 `box` 模拟变量（例如，研究new-loc函数，看看怎么用变量而不是box实现它），所以语言的表达能力并没减少，尽管由于box使用不慎可能（和变量相比）导致更严重的别名问题。

作为回报，开发者得到一种有意义的类型：除非某个数据结构中包含ref，否则它就可以被认为是不可变的；ref的存在也提醒开发人员和程序（如编译器），底下的值可能会发生改变。比如说，如果b是box，程序员就应该知道，将 `(unbox b)` 绑定到v，然后用v替换程序中所有的 `(unbox b)` 是不明智的做法：原来程序总是去获取box的当前值，改了之后就变成访问原先的值了。（反过来，如果程序员需要某个时间的值，无论以后box怎么被赋值，那么就可以获取当前值，将其绑定，而不是老是去unbox。）

8.4 参数传递

我们当前实现的解释器中，对于每个函数调用，总是分配新地址用于存储参数。这意味着：

```
(let ([f (lambda (x) (set! x 3))])
  (let ([y 5])
    (begin
      (f y)
      y)))
```

会计算得到5而不是3。这是因为，形参x的值和实参y的值存放在不同的地址，所以对x赋值不会影响y。

现在，试想程序以下面说的这种方式执行。当实参为变量时——它在内存中在有个地址——我们不再为该值重新分配地址，而是直接使用变量原来的地址。于是现在形参和实参指向的是内存中的同一块地址：即它们为变量别名（variable aliases）。这样对形参的赋值会影响调用者；上面的例子将计算得到3而不是5。这被称为传引用调用（call-by-reference）参数传递策略。

> 相反，我们的解释器实现了传值调用（call-by-value），Java等语言也采取这种参数传递策略。一个有点费解之处是，如果传递的值本身是可变的（译注：类似于我们的box），在被调用函数中进行的修改能被调用者看到。这仅仅是可变数据的产物，而不是传递策略导致的。请区分清楚！

在一段时间里，传引用调用被认为是好主意。使用它可以写出一些有用的抽象，比如swap函数，调用该函数将交换调用者手上两个变量的值。不过，这种特性的劣势远大于其优势：

- 粗心的程序员可能会无意间创建了别名变量，然后修改其值（而没有意识到自己这么做了），调用者可能永远不会注意到这种错误，直到某个特别条件触发了该修改。
- 有些人认为这种策略效率更高所以是必然的选择：他们如果不是传引用的话，其他策略需要拷贝大量数据。但是，传值调用也可以仅传递数据结构的地址。仅在这种情况（a并且b并且c）下需要拷贝数据：(a)数据结构是可变的，(b)不希望被调用者（译注，原文此处为调用者，逻辑关系并不合理故如此翻译）改变参数的值，(c)语言本身没有提供符号支持或者其他机制将此参数标记为不可变。
- 它必然会导致不统一的、非模块化的推理。例如，考虑如下的子程序：

  ```
  (define (f g)
    (let ([x 10])
      (begin
        (g x)
        ...)))
  ```

  如果允许传引用参数传递的话，程序员将不能仅看局部代码——也就只看这一段——就确定省略号中 `x` 的值。

如果某个语言非要允许传引用调用的话，至少需要让调用者决定是传引用——让在被调用者内部共享传入的内存地址——还是不使用传引用。然而即使使用这种方式也不怎么样，因为现在被调用者面临对称的问题——它的参数是不是个别名呢。传统的顺序式程序中，这还不是个问题，但是如果子程序是可重入的，被调用者就面临这种窘境。

所以是时候考虑一下引入任何这种东西是否值得了。如果调用者想要某个子程序执行某种赋值操作，传`box`值就好了。`box`表明，调用者接受——甚至说请求——被调用者进行赋值操作，执行结束后调用者只需从box中抽取出值。当然这样我们就不能写出很简洁的`swap`子程序，但是为了真实世界软件工程的考虑，这点小代价还是花的起的。

9 递归和循环：子程序与数据

递归指的是自我引用的行为。编程语言中存在（至少）两种形式的递归：数据的递归和控制的递归（程序行为，也就是函数的递归）。

9.1 递归与循环数据

数据中的递归还可以分两种情况：引用与自身相同类型的事物，或者就是直接引用自身。

第一种情况即我们传统称为递归数据。例如，树是一种递归数据结构：任一节点可以有若干子节点；每个子节点自身也是树。不过，如果编写程序遍历树，无需记录哪些节点已经被访问。树是有穷的数据结构。

与之对应的是图这种循环（cyclic）数据：节点引用其他节点，可能最终通过引用链引用回自身。（当然，节点还可以直接引用自身。）遍历图的时候，如果不记录已访问过的节点，计算过程就可能发散，即不会终止。图的算法需要记住已访问过的节点，从而避免重复遍历。

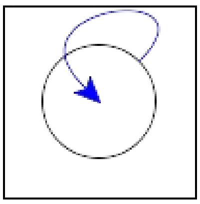
给我们的语言添加递归的数据结构，如链表或树，比较简单直接。主要需要实现两点：

1. 创建复合结构（compound structure）的能力（例如节点可以引用子树）；
2. 结束递归的能力（如树结构的叶节点）

练习

> 给语言添加内建数据类型：链表、二叉树

添加循环数据更为微妙。考虑循环数据的最简单形式，指向自身的单元格：



试试在Racket中定义它。尝试：

```
(let ([b b])
  b)
```

但这行不通：let中右边那个 `b` 未绑定。把语法糖解开可以看的更清楚：

```
((lambda (b)
   b)
  b)
```

为了清楚起见，我们可以重命名函数中的`b`：

```
((lambda (x)
   x)
  b)
```

明显`b`未绑定。

不使用额外的Racket构造的情况下【注释】，显然我们无法直接创建循环数据。我们需要给数据创建“地址”，然后在该地址中引用自己。注意这里是用了“然后”，它暗示时间的概念，即我们需要使用赋值操作。这样的话，我们可以试试用`box`来实现。

> 指`shared`构造，不过其他语言基本上都没有这个机制，所以这里我们也不深究它了。我们这里学习的东西正是`shared`幕后实现的基本原理。

计划如下：首先，创建box并将其绑定到某个标识符，设为`b`；然后改变box中的值，我们希望其中存什么呢？当然是对自身的引用。怎么获得该引用呢？通过名字`b`。通过这种方式，我们创建了环状数据：

```
(let ([b (box 'dummy)])
  (begin
    (set-box! b b)
    b))
```

注意，上面的程序在Typed PLAI中无法运行，后面会谈到如何给该程序添加类型。现在，要运行上面的程序，请使用动态类型的（`#lang plai`）语言。

运行上面的程序，Racket显示`#0='#&#0#`。这个表达式正是我们想要的。回想一下，前面提到过`#&`是Racket中box的显示方式。`#0=`（其中0换成其他数也是一样）是Racket中对于循环数据的命名方式。因此，上面结果字面意思就是“`#0`被绑定到了一个box，其内容为`#0#`，即绑定到`#0`的东西，即它自己”。

练习

> 在你自己的解释器中运行与这段代码，确保其产生循环的数据值。怎么检测这一点呢？

上述思想可以用于其它数据类型。通过这种方式，我们能够创建循环的链表、图等等。核心思想就是分两步做：先命名一个空的占位符；然后修改占位符中的内容为其自身；要获取“自身”，使用第一步中绑定的名字即可。当然，不限于“自循环”：我们也可以创建相互循环的数据（没有某个元素是循环的，但它们的组合是循环的）。

9.2 递归函数

澄清一下名词，递归函数不是引用与自身相同类型的函数，而是引用自身的函数。首先我们的语言需要已经添加了条件分支的特性（比如说前面第五章中，添加了条件指令判断是否是0），这样我们才能写出有意思的程序。

首先，用递归实现阶乘函数：

```
(let ([fact (lambda (n)
              (if0 n
                   1
                   (* n (fact (- n 1)))))])
  (fact 10))
```

这根本行不通！它将报错内层的 `fact` 未绑定，和前面循环数据的例子相同。

出现这种错误我们并不感到奇怪。毕竟到目前为止，我们实现的绑定机制并不会自动使函数定义支持循环（事实上，在一些早期的编程语言中，函数也不自动支持循环：递归被当作特殊的特性）。想要递归的话——即某个函数定义可以循环的引用自己——我们必须手工实现这点。

> 如果按惯例在顶层定义函数，你就不会遇到问题。顶层的绑定意味着它要么是变量，要么是box。所以下面说的模式基本上自动就帮你完成了。这也是为什么当你需要局部循环引用的时候，必须使用 `letrec` 或者 `local` ，而不是 `let` 的原因。

那么解决手段也很明了：问题和上一个类似，方案就也用一样的。还是三步走：先创建占位符，然后在需要循环引用的地方使用该占位符，最后在使用之前要对占位符赋值：

```
(let ([fact (box 'dummy)])
  (let ([fact-fun
         (lambda (n)
           (if (zero? n)
               1
               (* n ((unbox fact) (- n 1)))))])
    (begin
      (set-box! fact fact-fun)
      ((unbox fact) 10))))
```

事实上，我们并不需要 `fact-fun` ：这样写只是为了清晰起见。注意到 `fact-fun` 不是递归的，而且可以认为它是标识符而不是变量，所以我们可以直接使用它的值：

```
(let ([fact (box 'dummy)])
  (begin
    (set-box! fact
              (lambda (n)
                (if (zero? n)
                    1
                    (* n ((unbox fact) (- n 1))))))
    ((unbox fact) 10)))
```

这里有点小瑕疵，我们使用 `fact` 的时候总得 `unbox` 。如果语言中有变量，这么实现看起来更完美：

```
(let ([fact 'dummy])
    (begin
      (set! fact
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (- n 1))))))
      (fact 10)))
```

> 事实上，变量的一个用途就是简化上述模式的去语法糖过程，不再需要每次使用循环绑定的标识符时都得unbox。另一方面，通过一些额外的努力，去语法糖过程也可以把unbox带掉。

9.3 草率的观察

到这里我们发现一个遵从同样时间顺序的模式：创建、更新、使用。我们可以将这个过程裹在语法糖中。考虑实现下面的语法：

```
(rec name value body)
```

举个例子：

```
(rec fact
     (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
     (fact 10))
```

它将计算得到10的阶乘。该语法糖解开会得到：

```
(let ([name (box 'dummy)])
  (begin
    (set-box! name value)
     body))
```

这里，我们假设 `value` 和 `body` 中所有对 `name` 的引用都被改写为 `(unbox name)` ；或者换种方法，我们也可以使用变量：

```
(let ([name 'dummy])
  (begin
    (set! name value)
    body))
```

这自然就导致一个问题：如果我们搞砸了顺序呢？最有意思的是，如果我们在更新 `name` 到实际值之前使用它呢？那么我们将看到初始化时系统给该结构的无意义值，也就是原始形式的占位符。

最简单可以描述此问题的例子是：

```
(letrec ([x x])
  x)
```

或者等价的：

```
(local ([define x x])
  x)
```

在大多数Racket变体中，这会泄露占位符的初始值——这个值并没打算给大家使用。麻烦的地方是，这又是个合法的值，这意味着它至少可以被用于一些计算中。然而，如果无意中访问和使用它，那么后续的计算就是瞎扯。

这个问题通常有三种解决方案：

1. 确保该值足够模糊，以至于无法在有意义的上下文中使用该值。这意味着像`0`这种值就不能用，事实上语言中绝大多数数据类型都不该用。取而代之，语言应该创建一种新类型的值专作此用。将该值传入其它任何操作都将导致错误的抛出。 2. 对于任意一处标识符的使用，明确地检查其值是否是这个特殊的“过早”值。虽然这在技术上是可行的，但它会对程序造成了巨大的性能损失。因此，通常只有教学语言这么做。
2. 只允许递归构造用于绑定函数中，而且要求该绑定的右项必须在语法上是函数。不幸的是，这个解决方案过于激进，比如说它不允许了我们写出图这样的结构。

9.4 不用到显式的状态

聪明的你可能想到，还有一种方法可以定义递归函数（递归数据也是一样），而无需用到显式的赋值操作。

思考题

> 你应该已经明白，当我们使用`let`来定义递归函数时出了什么问题。请再试试。提示：需要更多的替换。不够再加，加满！

仅使用函数（字面意思上）获得递归是个了不起的想法。Daniel P. Friedman和 Matthias Felleisen在《The Little Schemer》一书中很好的描述了其做法。你可以读一下其在线样章。

练习

> 这个方案中用到了状态吗？有没有间接的用到呢？

10 对象

一门语言将函数作为值，就最为自然地提供了表示计算的最小单位。假设程序员需要把某个函数 `f` 参数化。任何语言都会允许把被动的数据——比如数字和字符串——用作函数参数。但是如果主动的数据——可以计算出结果的数据，比如说响应某种信息——也可以用作参数，这个想法就很有吸引力了。此外，作为参数传给 `f` 的函数——假设它遵从词法作用域——可以使用它的调用者提供的数据，而这些数据无需暴露给 `f`，这给安全和隐私提供了基石。正因如此，遵从词法作用域的函数成了设计很多安全编程技术的核心。

函数是好的东西，但是它太过简洁。有时候我们希望多个函数闭合于同一份共享的数据；共享的意义在于，当这份数据被其中某个函数修改时，我们希望其他函数能够看到修改后的结果。在这种情况下，不可能仅仅发送一个函数作为参数；发送一组函数更有用。接收方则需要能够从这组函数中提取出各个函数。这么一组函数，外加从中选取函数的方法，便是对象（object）的精髓。我们已经学过了函数（第七章）和可变结构（第八章），现在正是学习对象的最佳时机——同时前面学习的递归（第九章）也将派上用场。

我们来把此概念的对象添加到自己的语言中。然后我们将不断改进和扩展它，从而探究关于对象系统设计的各种维度。首先展示一下怎么将对象加入到核心语言中，但是由于想要快速构建许多不同的想法，我们很快就会转向基于去语法糖的策略。使用哪种方式取决于你是否认为理解它们对理解你的语言的本质至关重要。判断这点的一种方法是，看你的去语法糖过程变得有多复杂，以及，在给核心语言添加一些关键特性后，去语法糖的复杂度能否大幅降低。

> 我不能指望这里能讨论关于对象系统的一切，你可以阅读Éric Tanter的《面对对象编程语言：应用和解释》 来了解更多细节以及我们没有涉及到的主题。

10.1 不支持继承的对象

最简单的对象概念——可能是唯一所有谈论对象的人都能认同的定义——对象是：

- 值，
- 够将一些名字映射成
- 其它东西：值或者“方法（methods）”

从简约的角度来看，方法似乎就是函数，由于我们的语言已经实现了函数，我们先忽略它们之间的区别。

> 之后我们会发现“方法”和函数极其相似，但是在某些重要的方面有所不同：调用方式，还有其内部所绑定的东西。

10.1.1 核心语言中的对象

让我们往支持一等函数的核心语言（译注，即第七章中实现的语言）中加入简单的对象。显然我们必须扩展值的概念：

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)]
  [objV (ns : (listof symbol)) (vs : (listof Value))])
```

还要扩展语法，支持对象的构造表达式：

```
[objC (ns : (listof symbol)) (es : (listof ExprC))]
```

> 这里语言的设计中已做了一个抉择。在某些语言（如JavaScript）中，程序员可以直接写出对象。这是个非常受欢迎的概念，JavaScript中该功能的部分语法成了网络标准——JSON。在其他语言（如Java）中，对象只能通过调用某个类的构造函数来创建。这两种设计我们都可以模拟。要模拟后一种语言模型，我们必须遵从后文讨论到去语法糖中提出的程式化惯例，只在特定位置直接写出对象。

对这个对象表达式的求值很简单：对每个表达式位置都求值就行：

```
[objC (ns es) (objV ns (map (lambda (e)
                              (interp e env))
                            es))]
```

不幸的是，我们无法实际使用对象，因为无法获取其内容。为此，我们添加一个操作来提取成员：

```
[msgC (o : ExprC) (n : symbol)]  ;消息，核心语言
```

其行为就是直接：

```
[msgC (o n) (lookup-msg n (interp o env))]
```

练习

> 实现函数
>
> ```
> ; lookup-msg : symbol * Value -> Value
> ```
>
> 第二个参数的类型应该是`objV`。

原则上，`msgC`可以被用于获取任意类型的成员，但是简单起见，我们假设成员中只有函数。要使用某个成员，需要给其传入参数值。在核心语言的语法中这么写有点笨拙，所以我们假设去语法糖过程降低了语法复杂性：表层语法中消息调用同时提供了消息名和参数：

```
[msgS (o : ExprS) (n : symbol) (a : ExprS)]  ;消息，表层语言
```

去语法糖得 `msgC` 和函数调用：

```
[msgS (o n a) (appC (msgC (desugar o) n) (desugar a))]
```

至此，一个包含对象的语言就诞生了。例如，下面是对象定义和调用：

```
(letS 'o (objS (list 'add1 'sub1)
               (list (lamS 'x (plusS (idS 'x) (numS 1)))
                     (lamS 'x (plusS (idS 'x) (numS -1)))))
      (msgS (idS 'o) 'add1 (numS 3)))
```

它计算得 `(numV 4)` 。

10.1.2 通过去语法糖实现对象

在语言核心中定义对象也许是值得的，但是对于学习它来说这么做太麻烦了。替代方案是我们直接使用Racket语言中那些我们的解释器已经实现过的特性来表示对象。也就是说，假设我们看到的是去语法糖后的结果。（基于这个理由，我们会使用程式化的代码，可能某些表达式看上去并不必要，但请注意这是程序生成器输出的代码。）

注意：后面所有的代码都使用 `#lang plai` ，而不是 `typed` （静态类型）语言。

练习

> 为什么使用 `#lang plai` ？不然的话，在运行后面的代码的时候会碰到什么问题？这些问题好解决吗，比如引入新的数据结构来保证代码类型正确？如果简化我们的模型呢，比如让方法只接受一个参数？或者其中有些问题很难解决？

10.1.3 对象作为名称集合

首先实现我们之前实现的对象语言。对象是对给定名称进行分派的一种值。简单起见，我们用 `lambda` 表示对象，用 `case` 实现分派：

```
(define o-1
  (lambda (m)
    (case m
      [(add1) (lambda (x) (+ x 1))]
      [(sub1) (lambda (x) (- x 1))])))
```

> 注意到这个简单对象的实现是泛化了的 `lambda` ，带有多个“入口点”。相反， `lambda` 可以理解为只有一个入口点的对象，也因此它不需要“方法名”。

这和本章前面的定义的对象相同，使用其方法的方式也相同：

```
(test ((o-1 'add1) 5) 6)  ;;这个测试会通过
```

当然，这种嵌套的函数调用有点臃肿（并且将变得更加臃肿），所以我们最好提供一种方便的语法来调用方法——和前文 `msgS` 一样，不过我们可以简单将其定义为函数：

```
(define (msg o m . a)
  (apply (o m) a))
```

> 这里使用了Racket的可变参数目数语法：`. a` 的意思是，将剩下所有参数——零个或多个——绑定到名为 `a` 的链表。`apply` 将链表中的值取出作为参数来进行函数调用。

这样我们的测试就可以这么写：

```
(test (msg o-1 'add1 5) 6)
```

思考题

> 换用去语法糖的方式后，有些重大改变。你意识到是什么吗？

回忆一下之前定义的语法：

```
[msgC (o : ExprC) (n : symbol)]
```

注意到消息“名字”的位置必须是符号。即程序员在该位置必须字面写上符号。而在去语法糖的版本中，名字的位置只是表达式，当然该表达式必须计算得到符号；例如，可以这么写：

```
(test ((o-1 (string->symbol "add1")) 5) 6)  ;;这也会通过
```

这是去语法糖的常见问题：目标语言中有些表达式可能在源码中没有对应的表示，于是它们不能映射回去。幸运的是，通常我们不需要进行反向映射，不过某些调试和程序分析工具中可能需要这么做。重要的是，我们必须保证目标语言中不会出现无法在源码中对应的值。

有了基本的对象实现，接下来我们添加那些大多数对象系统中都有的特性。

10.1.4 构造器

构造器就是在对象构造时调用的函数。我们还没定义过这种函数。只要将对象从字面值转换成接受构造参数的函数，便可以达到效果：

```
(define (o-constr-1 x)
  (lambda (m)
    (case m
      [(addX) (lambda (y) (+ x y))])))

(test (msg (o-constr-1 5) 'addX 3) 8)
(test (msg (o-constr-1 2) 'addX 3) 5)
```

在第一个例子中，我们传入5作为构造器的参数，所以加3得8。第二个例子是类似的，这表明构造器的两次调用不会相互干扰。

10.1.5 状态

许多人认为对象的主要目的就是用来封装状态。【注释】我们当然保有这种能力。如果去除语法糖后的语言支持变量（当然支持 `box` 也行，代价是去语法糖过程会更麻烦些），我们很容易实现多个方法对同一个状态赋值，例如修改构造参数：

```
(define (o-state-1 count)
  (lambda (m)
    (case m
      [(inc) (lambda () (set! count (+ count 1)))]
      [(dec) (lambda () (set! count (- count 1)))]
      [(get) (lambda () count)])))
```

> Alan Kay——因发明Smalltalk和现代对象技术而获得图灵奖——不同意这一观点。在《Smalltalk的早期历史》中，他说，“在Smalltalk的早期历史中，往小了说（面向对象编程的动机）是寻找更易用的赋值，进一步则是尝试完全消除赋值。他补充说：“不幸的是，今天所谓的‘面向对象程序设计’大部分都是新瓶装旧酒。很多程序都充满了‘赋值式的’操作，只不过由更昂贵的附加子程序完成罢了。”

可以使用下面的代码序列测试：

```
(test (let ([o (o-state-1 5)])
        (begin (msg o 'inc)
               (msg o 'dec)
               (msg o 'get)))
      5)
```

请注意，对一个对象进行赋值不会影响到另一个对象：

```
(test (let ([o1 (o-state-1 3)]
            [o2 (o-state-1 3)])
        (begin (msg o1 'inc)
               (msg o1 'inc)
               (+ (msg o1 'get)
                  (msg o2 'get))))
      (+ 5 3))
```

10.1.6 私有成员

另一个常见的对象语言特性是私有成员：只在对象内部可见，外部就不可见。【注释】看上去这个特性还有待我们去实现，但我们已经有了局部作用域的、词法绑定的变量：

```
(define (o-state-2 init)
  (let ([count init])
    (lambda (m)
      (case m
        [(inc) (lambda () (set! count (+ count 1)))]
        [(dec) (lambda () (set! count (- count 1)))]
        [(get) (lambda () count)]))))
```

> 除此之外，在Java中，相同类型的其他类的实例也能访问“私有”成员。否则就没办法实现抽象数据类型了。

这么去除语法糖之后，不存在访问 `count` 的方法，词法作用域则确保它对外部不可见。

10.1.7 静态成员

对于对象的使用者来说，另一个有用的特性是静态成员：所有“相同”类型对象实例共享的成员。【注释】实际上，这就是（私有的）词法范围标识符，并且位于构造函数之外（这使其对所有构造函数的调用来说都是共享的）：

```
(define o-static-1
  (let ([counter 0])
    (lambda (amount)
      (begin
        (set! counter (+ 1 counter))
        (lambda (m)
          (case m
            [(inc) (lambda (n) (set! amount (+ amount n)))]
            [(dec) (lambda (n) (set! amount (- amount n)))]
            [(get) (lambda () amount)]
            [(count) (lambda () counter)]))))))
```

> 这里用引号是因为，对象有许多“相同”的概念。太多了。

我们把增加 `counter` 的那行放在该对象的“构造器”所在的位置，尽管它也可以在方法内部被操纵。。

测试就是构造多个对象，并确保它们每一个都影响了全局的 `count` ：

```
(test (let ([o (o-static-1 1000)])
        (msg o 'count))
      1)

(test (let ([o (o-static-1 0)])
        (msg o 'count))
      2)
```

10.1.8 带自引用的对象

到目前为止，我们的对象还只是打包的实名函数；或者你可以这么说，有多个实名入口点的函数。可以看到，很多对象系统中被认为很重要的特性可以通过函数和作用域实现，事实上很长一段时间里懂得 `lambda` 的程序员的确是这么做的，只是没有给这种做法起名字罢了。

对象系统一个不同与众不同的特征是，每个对象都自带了对该对象自己的引用，通常称为 `self` 或者 `this` 。【注释】我们可以方便的实现这一点吗？

> 对象的倡导者们经常采用的拟人化的术语“了解自己”，而我更喜欢这种略显枯燥的描述。事实上，请注意，我们无需要求助于拟人化，已经描述了很多对象系统的属性了。

10.1.8.1 使用赋值实现自引用

是的，可以这么实现，之前实现递归的时候我们已经见过此模式了；只需要将其一般化，引用对象自身而不是 `box` 或者函数：

```
(define o-self!
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(first) (lambda (x) (msg self 'second (+ x 1)))]
                [(second) (lambda (x) (+ x 1))])))
      self)))
```

可以看见这就是递归的模式（递归函数），稍作调整。在方法 `first` 中使用自引用调用了方法 `second` 。测试表明这么做可行：

```
(test (msg o-self! 'first 5) 7)
```

10.1.8.2 不用赋值实现自引用

如果你研究过怎么不使用赋值实现递归，那么你会发现该方案也适用于这里。

```
(define o-self-no!
  (lambda (m)
    (case m
      [(first) (lambda (self x) (msg/self self 'second (+ x 1)))]
      [(second) (lambda (self x) (+ x 1))])))
```

现在每个方法需要传入 `self` 参数。这意味着方法调用也需要修改，以遵循新模式：

```
(define (msg/self o m . a)
  (apply (o m) o a))
```

也就是说，当调用对象 `o` 的方法时，必须把 `o` 作为参数传递给方法。显然这种方式存在隐患，调用方法的时候可以传入不同的对象作为 `self` 。因此将这个功能提供给程序员可能是个坏主意；如果使用这种技术，则只能通过去语法糖来实现。

> 尽管如此，Python还是在其表层语法中这么做了。尽管这种致敬Y-combinator的行为令人感动，但是由此带来的脆弱性也许不必要。

10.1.9 动态分发

最后，我们希望我们的对象可以处理对象系统的这个特性，调用者可以进行方法调用，而无需知道或者决定哪个对象会处理该调用。假设我们有个二叉树数据结构，树中要么是不含值的节点或者含值的叶节点（译注：原文如此，和后面的代码有相反之处）。传统的函数中，我们需要借助某种形式的条件判断—— `cond` 、 `type-case` 、模式匹配，或与之等价的东西——穷举不同形式的树并根据对应形式来选择执行。如果树的定义扩展了，包含了新的类型，那么所有相应的代码段必须修改。动态分发（dynamic dispatch）将该条件选择移到语言内部，使得用户程序可以不用处理这种情况，从而解决此问题。它提供的关键特性是可扩展的条件。这也是对象提供的可扩展性的一个方面。

> 动态分发使得系统具有黑盒可扩展性，因为系统的某个部分可以在不触及其他部分（代码修改）的情况下扩展，这个属性也被认为是面向对象编程的一大好处。这的确是对象相比函数的优势，然而函数相比对象有个对等的优势，事实上很多对象程序员使用访问者模式（Visitor pattern）来组织代码，使其看起来更像函数式的。请参阅Synthesizing Object-Oriented and Functional Design to Promote Re-Use ，其中包括具体的例子，给出此问题的完整描述。试着用你最喜欢的语言解决这个问题，然后可以看看Racket中的解决方案。

先来定义两种类型的树对象：

```
(define (mt)
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(add) (lambda () 0)])))
      self)))

(define (node v l r)
  (let ([self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(add) (lambda () (+ v
                                     (msg l 'add)
                                     (msg r 'add)))])))
      self)))
```

于是，我们可以构造具体的树：

```
(define a-tree
  (node 10
        (node 5 (mt) (mt))
        (node 15 (node 6 (mt) (mt)) (mt))))
```

最后，测试一下：

```
(test (msg a-tree 'add) (+ 10 5 15 6))
```

注意到，在测试案例中，还有在`node`的`add`方法中，都调用了`add`方法而没有检查接收方是`mt`还是`node`。运行时系统提取出接收方的`add`方法并执行。用户的程序中没有条件表达式，这正是动态分发的精髓。

10.2 成员访问的设计空间

对于成员名称的处理我们已经有两个正交的纬度。一个维度是名字是静态给定还是计算给出的，另一纬度是名字的集合是固定的还是可变的：

	名字是静态的	名字是计算求得的
成员固定	基本的Java	Java中通过反射计算出的名字
成员可变	无法想象	大部分脚本语言

只有一种情况毫无意义：如果强制程序员在源码中显式指定成员名，那么就无法添加新的可访问的成员了（当然，访问曾经存在过但是被删除的成员还是会报错）。其它的几种情况都已经在各种语言中被尝试过了。

右下方那种情况密切对应于那些使用哈希表表示对象的语言。成员名字即哈希表的索引。一些语言将这种风格推到极限，当索引是数字时也同样处理，于是对象和和字典（甚至数组）都混到了一起。即使对象只处理“成员名字”，这种风格的对象也给类型检查带来极大困难，这可不是什么好事。

因此，本章的其余部分，我们将坚持使用“传统的”对象，成员固定，甚至会让它的名字只能是静态的（对应于左上角那种）。即使这样，我们将发现仍有很多待学习的东西。

10.3 还有点啥（else中放什么）？

截至目前，我们的`case`表达式并不包含`else`子句。这么做的一个原因是，方便使得我们的成员（及成员数量）可变；尽管前面我们也讨论过，使用其它方式实现，例如哈希表，可能是更好的选择。相反，假如对象成员固定，把对象去语法糖实现为条件表达式从演示的角度来讲很合理（因为这种实现方式强调了成员名称固定这一点，而哈希表实现就将这一点交给了解释器，这么做容易导致错误）。不过，还有一个很好的原因，需要用上`else`子句：继承（inheritance）。它指的是，将控制“链式地”交给另一个对象，称为父对象。

还是从前文中去语法糖对象模型开始。为了实现继承，需要提供给对象“某种东西”，当遇到其识别不了的方法，委托它实现。“某种东西”怎么选择将导致迥异的设计结果。

一种简单的选择，另一个对象。

```
(case m
  ...
  [else (parent-object m)])
```

基于我们的实现，这么做的话，我们将在父对象中搜索当前对象中不存在的方法（并且递归的搜索父对象的父对象）。如果找到与名称对应的方法，那么方法就会链式的返回最初的 `msg` 调用。如果找不到方法，最后那个对象可以报错“未找到消息”。

练习

> 注意到调用 `(parent-object m)` 就像“半个 `msg` ”一样，和左值是“半个查找”类似。两者有什么联系吗？

让我们来试试这个想法，扩展我们的树实现另一方法 `size` 。我们通过给对象 `node` 和 `mt` 分别实现“扩展”（你可能想叫它“子类”，但现在请先忍住）的方式实现，也就是使用前述的模式。

> 这里不会对现有的定义做任何编辑，这正是对象继承的意义所在：以黑盒的形式重用代码。这意味着，彼此不认识的各方可以各自扩展相同的基本代码。如果他们必须编辑基本代码，首先他们必须知道对方的修改，此外，某一方可能不喜欢另一方做的编辑。继承就可以完全避开这种麻烦。

10.3.1 类

我们立刻就遇到了难题。构造器的模式是这样的吗？

```
(define (node/size parent-object v l r)
  ...)
```

这段代码表明，父对象和对象构造器的其他参数处于“同一级别”。这看上去很合理，只要所有这些参数都给定了，该对象也就被“完全定义”了。然而，我们的代码中还有：

```
(define (node v l r)
  ...)
```

我们需要把所有的参数写两遍吗？（当有什么相同的东西需要写两次，应该考虑一下我们是不是有啥地方没有保持一致，因此引入了微妙的错误。）以下是替代方案：**`node/size`** 可以构造其父对象的实例。也就是说，传给 `node/size` 指明父对象的参数不是父对象本身，而是父对象的构造函数：

```
(define (node/size parent-maker v l r)
  (let ([parent-object (parent-maker v l r)]
        [self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(size) (lambda () (+ 1
                                     (msg l 'size)
                                     (msg r 'size)))]
                [else (parent-object m)])))
      self)))

(define (mt/size parent-maker)
  (let ([parent-object (parent-maker)]
        [self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(size) (lambda () 0)]
                [else (parent-object m)])))
      self)))
```

每次调用对象构造器的时候，就必须要记得传入父对象的构造函数：

```
(define a-tree/size
  (node/size node
             10
             (node/size node 5 (mt/size mt) (mt/size mt))
             (node/size node 15
                        (node/size node 6 (mt/size mt) (mt/size mt))
                        (mt/size mt))))
```

显然我们可以通过合适的语法糖简化上面这一堆东西。写两个测试来确保原功能和新加功能都正确：

```
(test (msg a-tree/size 'add) (+ 10 5 15 6))
(test (msg a-tree/size 'size) 4)
```

练习

> 把这段代码改写成self调用模式的，不使用赋值（第10.1.8.2节）。

这里展示的就是类（class）的精髓。给函数加上父参数后它就是......好吧，真的有点棘手。现在我们把它称为**blob**（难以名状的一团）。blob对应于Java程序员在编写类时定义的内容：

```
class NodeSize extends Node { ... }
```

思考题

> 那么，为什么我们不把它叫做“类”呢？

当程序员调用Java的类构造器时，它实际上构造了继承链上的所有对象（当然，编译器可能会对此优化，只需要进行一次构造器调用和一次对象分配）。每个父类都会对应创建一个私有的对象（对于静态方法来说是私有的）。问题是，这些对象中有多少是可见的。Java的选择和我们上述的实现不同，是对于每个给定名字（和签名）的方法只保留一个，不管该方法在继承链上被实现了多少次，而所有的字段都被保留，可以通过强制类型转换去访问。后者细想是合理的，因为对字段来说，可能会有一些基于它的不变量，所以保证它们彼此分离（因此所有字段都存在）是很有必要的。相比之下，很容易想出来一种方式可以使所有方法可用，而不仅是继承层次中最低（即最精炼）的方法。很多脚本语言采用这种方法。

练习

> 前面的代码犯了一个本质错误。`self`引用的是同一个语法上的对象，而它需要引用的是最精炼（继承层次中最低）的对象：这个问题被称为开放式递归（open recursion）。【注释】修改对象的表示法，使得`self`总是引用对象最精炼的版本。提示：你会发现，self调用的方式（10.1.8.2节）更方便。
>
> 这展示了从传统对象获得的另一种可扩展性形式：可扩展递归（extensible recursion）。

10.3.2 原型

在前文的描述中，我们给每个类提供了其父类的描述。构造对象时将沿着继承链创建每个类的实例。关于父代还有一种想法：它不是需要实例化的类，而就是对象本身。这样拥有相同父代的子代都会看到同一个对象，这意味着从某个子对象中修改该对象内部状态将对其它子对象可见。该共有对象被称为原型（**prototype**）。

> 代表性的基于原型的语言是Self。虽然你可能听说过JavaScript是“基于”Self的，但是从其源头来研究这个想法是有意义的，而且Self展示了原型这个概念最纯粹的形式。

一些语言设计者认为原型比类更为基础，因为原型（外加语言中的其他基本机制，比如函数）可以实现类——但是反之则不行。前面我们基本上就是这么做的：每个“类”函数中都包含了对对象的描述，所以类就是返回对象的函数。如果我们假设这是两个不同的操作，直接继承对象，我们将得到类似原型的东西。

练习

> 修改继承模式，实现类似Self的、基于原型的语言，而不是基于类的语言。因为类为每个对象提供其父对象的不同拷贝，所以基于原型的语言可以提供克隆操作，从而简化在原型上模拟类的操作。

10.3.3 多重继承

你可能会想到，为什么（方法在本对象中找不到时）只提供一个选项呢？很容易把这个推广到多个选项的情况，这也很自然的导出多重继承（multiple inheritance）。有多个父辈之后很显然的问题是，查找方法时按照何种顺序进行。继承关系组织成树状结构，糟糕的是，并没有权威的顺序可供使用：比如是深度优先呢还是广度优先呢（两种做法都能找到论据支持）。更糟糕的是，例如blob A扩展自B和C；而B和C都扩展自D。【注释】问题来了：A的实例中包含一个还是两个D对象呢？只包含一个既节省空间且行为可能更符合期望，那么，访问该对象时是访问一次还是两次呢？两次访问之间应该没有什么区别，所以似乎没有必要。但一次访问意味着B或C之一的行为可能会改变。诸如此类。结果，几乎每一个支持多重继承的语言都伴随着一个微妙的算法，仅仅是定义查找的顺序。

> 这就是臭名昭著的菱形继承（diamond inheritance）问题。如果你选择在语言中包含多重继承，关于这个问题涉及的设计抉择可能需要你纠结好长时间。你几乎不可能找到规范的解决方案，所以你的痛苦才刚刚开始。。

多重继承只有在你思考之前才有吸引力。

10.3.4 （高超的）Super

很多语言中支持super调用，即调用继承链上一层中的方法或者访问上一层中的字段。【注释】包括在对象构造的时候这样做，在那里通常需要调用所有的构造函数，以确保对象被正确定义。

> 注意这里说的是“链”。在多重继承的情况下这些概念要复杂的多。

我们已经对向“上”调用习以为常，也许我们忘了问这是否是最自然的方向。请记住，构造器和方法的任务是维护不变量。我们应该更信任谁，超类还是子类？有些人认为，子类更为精炼，所以它拥有关于对象最全面的描述。但反过来说，超类必须保护其不变量不受无知的子类胡乱篡改。

这是关于继承到底是什么的两种截然不同的认知。向上意味着我们认为扩展是要替代超类。向下意味着我们认为扩展是改善父代。通常我们将子类继承视为后者（改善和精炼），但是为什么我们的语言进行调用的时候却选择了“错误的”方向呢？因此，有些语言探索了默认向下调用。

> gbeta是一门由众多有趣特性的现代语言，它支持 inner（即向下调用）。考虑结合这两个方向也是非常有趣的。

10.3.5 Mixin和Trait

回过头讨论我们的“blob”。

在Java中当我们写下一个“类”时候，那对大括号中事实上是什么东西呢？它不是完整的类：完整的类取决父类，那又递归的取决于它的父类。其实，我们在大括号内定义的是类的扩展。仅当在这个定义中加入父类后，它才是个完整的类。

自然我们要问：为什么？为什么不把扩展的定义和将扩展应用于基类这两个行为分开呢？即，将这段代码：

```
class C extends B { ... }
```

分割成：

```
classext E { ... }
```

和

```
class C = E(B)
```

其中 `B` 是某个定义好的类。

看上去这样好像只是用更长的代码实现一样的东西。但是这种类似函数调用的语法不禁让我们浮想联翩：可以将某个扩展“应用”于多个不同的基类。比如说：

```
class C1 = E(B1);
class C2 = E(B2);
// ...
```

诸如此类。通过将E的定义和其扩展的类分离开，我们将扩展从固定基类的暴政中解放出来。这种扩展有个名字：**mixin**。

> “mixin”一词起源于Common Lisp，是多重继承的特定使用模式。鸡窝里飞出金凤凰。

Mixin使得类定义具有更好的组合性。它提供了很多多重继承的好处（重用多段功能代码），但是避免了多重继承的麻烦（例如没有前面讨论的复杂的查询顺序问题）。采用去语法糖的方式的话，mixin还非常容易实现。Mixin基本上就是“类的函数”。我们的目标语言支持函数，而且已经确定了类去语法糖后的表达式，该表达式可以放入函数中，这意味着实现简单的mixin模型非常容易。

> 这里的情况是，去除语法糖后的目标语言拥有良好的通用性，如果我们将其映射回源码语言，就能获得更好的结构。

在静态类型语言中，好的mixin设计完全可以改善面向对象编程的实践。假设我们要定义一个基于mixin的 Java。如果mixin等效于类到类的函数，那么这个“函数”的“类型”是什么？显然，mixin应该使用接口（interface）来描述其输入和输出。Java支持后者（但不强制要求），但是不支持前者：类（的扩展）扩展的是另一个类——这个类中所有的成员对扩展都是可见的——而不是其接口。这意味着子类获取了父类所有的行为，而不是其规范。如果修改父类，就有可能导致子类出错。

在支持mixin的语言中，我们就可以这么写：

```
mixin M extends I { ... }
```

其中I是接口。这样M可以用来扩展实现了接口I的类，语言能保证只有I中指定的成员在M中可见。这就遵循了好的软件设计的重要原则之一。

> “面向接口编程，而不是面向实现（Program to an interface, not an implementation）”
> —— 《设计模式》

好的mixin设计还可以更进一步。按照定义，一个类在继承链中只能使用一次（如果某个类的引用它自己，那么继承链上势必存在环路，这会导致无限循环）。反之，当我们编写函数时，就不会有这种顾虑（例如：`(map ... (filter (map ...)))`）。使用某个mixin两次有意义吗？

> 当然有！请参阅Classes and Mixins 的第3和第4节。

mixin解决了库设计中出现的一个重要问题。假设我们有十几个不同的特性可以用不同的方式进行组合，我们应该提供多少个类？更甚之，并不是所有特性都可以相互组合。显然，产生所有组合对应的类不现实。更好的方案是允许程序员选择他们关心的特性，且提供必要的机制防止不合理的组合。这正是mixin所解决的问题：mixin提供类的扩展，程序员可以自行组合，而接口必须要能对上，从而创建自己需要的类。

> Racket的GUI库中广泛使用了mixin。例如`color:text-mixin`的输入是基本的文本编辑器接口，输出是彩色的文本编辑器接口。后者本身也是一种基本的文本编辑器接口，于是其他基本文本相关的mixin还可以继续应用于其输出。

练习

> 你最喜欢的面向对象语言的库是怎么解决上述问题的？

Mixin也有局限：只能进行线性的组合。这种限制有时会给程序员带来不必要的负担。将mixin泛化，不是只对单个mixin扩展，而是扩展一组mixin，这被称为**trait**。当然，允许扩展多个就必须要处理潜在的名字冲突。因此实现trait必须同时提供解决名字冲突的机制，通常是某种名称组合代数。Trait是mixin的补充，程序员可以自行选择最满足其需求的机制。Racket支持mixin和trait。

11 内存管理

11.1 垃圾

垃圾（garbage）指的是已分配但是不再需要的内存。典型的编程语言的运行时系统采用两种不同的内存分配方式。一种是分配给环境；这种分配方式要和静态作用域保持一致，所以它只需要支持推入（push）和弹出（pop）操作。函数调用返回时，为其环境分配的空间也被返回，供后续函数使用，看似没有成本。【注释】与之相对，在贮存中分配的内存必须伴随某个值的一生，可能要超过其创建位置的作用域——事实上，它可能一直存活下去。因此，我们需要不同的策略来回收在贮存中分配空间所产生的垃圾。

> 并非没有成本。硬件必须执行“弹出”指令。这不见得就一定比其他内存管理策略更高效。

空间回收的方法有很多，大体可以分到两个阵营中：人工和自动。人工的方式依赖于开发者能够了解内存的使用，并正确的释放不需要的内存。一般认为，人并不擅长做这种事（虽然在某些情况下，人类拥有机器所无法获取的知识）。因此，几十年来，自动化的方法越来越普及。

11.2 什么样的垃圾回收是“正确的”？

垃圾回收既不应该太早地收回空间（可靠性，soundness）也不能太晚（完备性，completeness）。虽然两者都可以被视为缺陷，但是它们的影响并不是对称的：可以说，过早收回糟糕得多。这是因为，如果过早回收了某个贮存地址，计算将继续，并可能将其他数据写入该地址，从而访问到无意义的数据。往好了说，这会导致程序不正确，极端情况下后果更严重，比如可能会导致安全问题。反之，过迟收回会导致性能损失，并且可能最终导致程序终止，尽管此时存在理论上可用的内存。这种性能损失以及程序过早终止很令人讨厌，在某些关键任务系统中可能会导致重大问题，不过，至少程序不会进行无意义的运算。

理想情况下，我们希望拥有所有的这三项：自动化（automation），可靠性和完备性。然而，这里我们面对的是不可兼得的情形，最多只能选择两项。理想的人类能够做到可靠性和完备性，但实践中实现其中一个都很少见。【注释】计算机可以实现自动化，同时可以提供可靠性和完备性中的一个，但可计算性论证表明，自动化的计算过程不能同时达成这两者。实践中，自动化技术一般选择实现可靠性，出于以下原因：（a）它造成的损害最小；（b）它相对更容易实现；（c）在添加一些人工帮助的情况下，可以接近完备性。

> 你当然是完美的，但是你的程序员同行呢？顺便说一下，经济学理论在等你验证呢。

11.3 人工回收

人工的最彻底的方式是将所有内存回收交由人操作。例如，在C语言中提供了两个基本指令：`malloc`用于分配内存，`free`用于释放内存。`malloc`的输入是（内存的）大小，返回是对贮存的引用；`free`的输入是这种引用，释放其占用的内存。

```
        “在当代欧美语言，"Moloch"摩洛这个词有特定的引申义，指代需要极大牺牲的人物或者事业。”——
[维基百科，摩洛词条](http://en.wikipedia.org/wiki/Moloch)
        “我不认为这个名字听起来像malloc是巧合。”——Ian Barland
```

11.3.1 完全人工回收的代价

先来考虑一下这些操作的复杂度。首先我们假设`malloc`有个指向贮存的关联寄存器（比如`new-loc`），每次分配的时候直接获取下一个可用地址。这个模型非常简单——可惜只是看上去简单而已。问题出在当你需要用`free`释放内存时。如果调用`free`针对的是最后一次`malloc`分配的内存，那么没有问题；但是贮存中数据一般不遵堆栈的规律。如果释放的不是最新分配的内存，将会在贮存中留下空洞。空洞会导致碎片化（fragmentation），最坏的情况下，即使贮存中有足够的空间，也无法分配任何对象——许多分割的碎片，没有一个足够大。

练习

> 原则上，我们可以通过使所有空余空间相邻来解决碎片化的问题。怎么达成这一点？仔细考虑所有的后果，然后描述一下如何手工进行这项工作。

在大多数手动内存管理方案中，碎片化仍然是个不可克服的问题，不过在这个看上去很简单的方案里还有其他东西值得考虑。释放某个值之后会发生什么？运行时系统需要用某种方式记录这块内存可被分配。它是通过维护空闲表——空闲空间的链表——来达成这点的。稍作思考就会想到问题，空闲表存在哪，它的内存又由谁来管理呢？答案是空闲表存放在空闲的内存单元格中，这就意味着内存分配时存在最小分配单元。

那么，原则上，每次`malloc`现在必须遍历空闲表以找到合适的位置。说“合适”是因为分配者必须做出复杂的决定。遇到第一个匹配的空间就分配呢还是继续找找？而且“匹配”又是怎么定义的呢？应该选取那些大小刚好的空间，还是将大些的空间拆分成小块（从而增加创建不可用的小空间的可能性）？还有其它诸多问题。

程序员希望内存分配高效。【注释1】因此，实践中，分配系统倾向于只使用一组固定的尺寸，通常是2的幂。这样我们就可以不是只维护一个空闲表，而是为每个尺寸（都是2的幂）维护一个空闲表。然后再维护一个指向这些表的数组，位操作可以减小数组索引的代价。当然，这样会浪费一些空间，因为当需要那些不是2的幂尺寸的内存时，最终分配给其的内存尾部将会有空余。（这是计算机科学中经典的取舍(trade-off)：空间换时间）。`free`需要将释放的内存放到合适的链表中，有时候还需要将较大块的内存分割成小块以为将来的分配做准备。这个模型中的任何部分都不像看上去的那样高效。【注释2】

> 如果内存分配不够高效，开发者会尝试各种奇技赢巧来重用程序中的值，这会降低代码的清晰性，很有可能会导致错误。
>
> 特别地，`free` 并不免费（译注：双关）。

当然，所有这些都基于程序员可以写出可靠（忽略完备）程序的基础上。但是他们做不到。

11.3.2 引用计数

由于完全手工内存回收给程序员带来极大的负担，一些半自动化技术被广为使用，最为人知的便是引用计数（reference counting）。

使用引用计数的方式，每个值都关联一个计数，记录对其引用的个数。程序员负责负责递增和递减这些计数。当计数降为0时，该值的空间可以安全的回收供未来使用。

请注意，上面简单的定义中隐藏了两个重要假设：

1. 程序员可以记录每一次引用。回忆一下，别名也是引用。因此，当写出下面的代码时，

   ```
   (let ([x <some value>])
     (let ([y x])
       ..))
   ```

   程序员需要记住 `y` 是对 `x` 引用的那个值的第二次引用，因此要增加该值的引用计数。
2. 每个值只有有限个引用。如果数据中存在环路，这条假设不成立。

由于需要手动递增和递减引用，这种技术缺乏可靠性与完备性。事实上，上述第二个假设自然导致完备性的丧失，而第一个假设则指出了最简单的方式来打破可靠性。

手工管理内存的弊端还可以更为深层隐晦。由于程序员负责释放内存（或者，等效的，管理引用计数），内存管理策略必须成为每个库接口的一部分：即，“库中分配的值谁来释放？库会否释放传递给它的值？”很不幸，用文档准确记录、并遵守这种策略信息极其困难，更糟的是，它会导致文档中充斥关于底层的细节，它们通常与库要封装的行为毫无关系。

一个有趣的想法是将计数值的增减自动化。另一个想法是在实现中添加循环检测（cycle-detection）。引入这两者将解决上述的很多问题，但是引用计数还有一些其它问题：

- 引用计数会增加每个对象的大小。计数器需要足够大以防止溢出，又要足够小以避免过多的内存占用。
- 对这些计数器值的增减花费的时间会相当可观。
- 如果一个对象的引用计数降至0，那么它所引用的所有内容的计数值都需要减一，这种行为可能会是递归的。这意味着一次释放操作可能会花费大量时间，除非使用聪明的“惰性（lazy）”技巧（这样的话又会导致内存占用增加）。
- 为了减少计数值，我们需要遍历已经是垃圾的对象。这看上去很违反直觉：遍历我们已经不感兴趣的对象。工程实践中这会产生后果：这些我们不感兴趣的对象有可能已经很久没有被访问过了，这意味着它们可能被换页换出内存了。引用计数器需要将它们换页

回内存，仅为了告诉它们它们不再被需要了。

出于所有这些原因，应谨慎引用计数。你不应接受它作为默认，而是应该问自己，为什么拒绝通常被认为更好的自动化技术。

练习

> 如果引用计数溢出了，哪些正确性属性被破坏，是怎么被破坏的？权衡利弊。

11.4 自动回收，或垃圾收集

> 有些人认为引用计数是“垃圾收集”技术的一种。我更喜欢用后一个术语来指完全自动的技术。但是浏览网页时请注意可能的混淆。

现在让我们来简要地考察一下让语言的运行时系统自动化回收垃圾的过程。我们将使用缩写GC（**Garbage Collection**）同时指代垃圾回收的算法与垃圾回收的过程，上下文可以帮你区分具体指代哪个。

11.4.1 概览

所有GC算法的核心是通过值间引用关系遍历内存。遍历从根集（root set）开始，也就是是程序可能引用贮存中值的所有地方。通常，根集由环境中的绑定变量以及全局变量组成。在实际实现中，还需要考虑到类似寄存器中的引用这种易逝值。从根集开始，算法使用一系列算法——通常是深度优先搜索【注释】的变体——来遍历所有可访问的值，以识别所有存活的值（即，通过一些程序操作的序列可用到的值）。按定义所有其它数据就是垃圾。不同的算法使用不同的方式回收这些空间。

> 通常选用深度优先搜索，因为它适用于基于堆栈的实现。当然，你可能（也应该）想知道GC自己的栈存储在哪里！

11.4.2 事实和可证性

如果你仔细阅读的话，你会发现上面我描述了一个算法。这是实现的细节，而不是规范的一部分！垃圾回收的规范是事实（truth）的表述：我们要准确地回收所有是垃圾的值，不多也不少。但是对于任何图灵完备的编程语言，我们都没法得出这一事实，于是我们退而求其次，寻求可证性（provability）。上述的算法描述提供了存活性的有效“证明”，其补集就是垃圾。这个方案当然还有变种，收集更多或更少的垃圾，取决于证明“垃圾性”的不同强度。

上面的说的最后一点指出了严格规范术语描述中的缺陷，对于要回收多少垃圾它完全没有说明。考虑一下极端情况实际上是有益的。

思考题

> 定义一个可靠的垃圾回收策略很简单。同样，定义一个完备的的垃圾回收策略也非常简单。你能想到怎么做吗？

要做到可靠，我们只要确保不会错误的移除任何可能存活的数据。一种确保无疑的方式就是完全不回收垃圾。与之对应，完备的GC回收所有东西。显然这两者都是无用的（后者显然极其危险）。这为我们的工程实践指明了一点，我们不仅需要GC是可靠的，也希望它足够完备，同时还要足够高效。

11.4.3 核心假设

能够可靠地执行GC依赖于两条关键的假设。一条有关语言的实现，另一条有关语言的语义。

1. 对语言中的值，GC需要知道该值的类型以及它在内存中的表示法。例如，当遍历到`cons`单元，它必须知道：
   i. 这是一个`cons`单元；因此，
   ii. 它的`first`在哪里，例如位于4个字节的偏移量的地方，
   iii. 它的`rest`在哪里，例如位于8个字节的偏移量的地方。
   显然，这个属性必须递归地保持，使得遍历算法能够正确映射内存中的值。

2. 程序不能通过下面两种方式生成引用：
   i. 对象引用不能发生在语言实现预先定义的根集之外。
   ii. 对象引用只能指向对象中明确定义的点。
   违反第二条时，GC将完全乱套，错误的解释数据。第一条看上去显而易见，如果它被违反，意味着运行时系统错误地理解语言的语义。然而这条的后果有点微妙，下面将会讨论。

11.5 保守垃圾回收

上文说过，一般根集包含环境、全局变量和一些易逝值。引用还可能出现在什么地方？

在大部分语言中，没有其他地方了。但是有些语言（说的就是你们，C和C++）允许将引用转换成数，以及将任意数转换成引用。因此，原则上，程序中的任何数值（由于C和C++类型系统的特性，程序中几乎任何值）都可以被视为引用。

两个原因使得它问题重重。首先，GC不能只将其注意力集中到一个较小的根集；现在整个贮存都是潜在的根集。其次，如果GC试图以任何方式修改某个对象——例如在遍历时记录一个“访问”位——这时它可能修改了一个非引用值：例如，它可能实际上改变了程序中某个（看似无关的）数型常量。因此，像C和C++这样的语言中的特征组合起来，使得合理而有效的GC非常困难。

但并不是不可能。一个令人兴奋的研究方向——称为保守GC——成功的为此类语言创造了足够高效的GC系统。保守（conservative）GC背后的基本原则是，尽管理论上每个贮存地址都可能属于根集，但实际上它们大部分都不是。它会通过一系列聪明的观察来推断出哪些位置肯定不是引用（这点和传统GC相反），然后将它们安全地忽略掉：例如，在字节对齐的体系架构中，奇数值不可能为引用。通过忽略大部分贮存，通过对程序行为作出一些基本的假定（例如程序不可能产生某种类型的引用），并且小心操作不去修改贮存（例如，不改变值中的比特，不移动数据）的情况下，可以得到一个还算有效的GC策略。

> 刻鹄类鹜。

保守GC在那些使用或者依赖C和C++实现的编程语言中比较常见。例如，早期的Racket就完全依靠它。这是基于以下原因：

1. 它是种便捷的自举技术，语言实现者能得以将精力集中在其它更富革新性的特性上。
2. 如果语言能控制所有的引用（比如Racket），那么可以使用便于提高GC效率的内存表示法（例如，用1填充所有(真正的)数的最低有效位）。
3. 它使得该语言和C以及C++实现的库交互变得容易（当然前提是这些库也符合该技术的要求）。

这里需要解释一下名词。如前所述，所有实用的GC技术都是“保守的”，也就是说它们用（潜在的）可访问性代替真实中的是否访问。然而，“保守”这个词已经成为专门的术语，指在不合作（但不是故意对抗）的运行时系统中工作的GC技术。

11.6 精确垃圾回收

在传统的GC术语中，“保守”的反义词是精确（precise）。这也是误称，因为GC不可是精确的，即同时做到可靠和完备。这里精确更多是对识别引用能力的表述：当面对值时，精确GC知道什么是和不是引用，以及引用的位置在哪。相对保守GC，这省去了猜测哪些值不是引用（并以此尽可能多地消除潜在引用）这项繁重的工作。

大多数当代语言的运行时系统使用精确GC，而精确GC领域中存在大量的实现技术。我推荐Paul Wilson 的调查报告（虽然这份材料有点显老，但在这个快速发展的领域中仍是很好的资源）和Richard Jones 的书和资料。最后，对于世代垃圾收集器的概述，可以读一下简单的世代垃圾收集器和快速分配。

12 表示层抉择

回去看看我们将函数作为值的那个解释器，你能找到其中不一致的地方吗？

思考题

> 找到了吗？

考虑一下我们是怎么表示这两种值的：数和函数。忽略其外面 `numV` 和 `closV` 这一层，注意它们底层的数据表示。我们使用Racket中的数来表示要解释的语言中的数，但是我们没有使用Racket中的函数（闭包）来表示要解释的语言中的函数（闭包）。

这就是不一致的地方。更一致的做法是，要么都用Racket中的值表示，要么都不用。那么我们为什么要做出这种决定呢？

这么做是要说明一个问题。本章我们就讨论此问题。

12.1 改变表示

我们暂且探究一下数。Racket中数很强大所以我们重用它：它支持任意大小的整数（**bignum**）、有理数（这点受益于整数的bignum表示）、复数等等。因此，它能表示出大部分常规语言中的数系统。然而，这并不意味着它就是我们想要的：它可能过于简单或者过于复杂：

- 如果我们需要的是某种受限的数系统，它就过于复杂了。例如Java中规定了一组定长的数的表示（如：int被指定为32位的）。超出这个规定范围的数在Java中将不能直接被表示，同时算术运算也遵循此范围（例如：由于溢出，1加2147483647将不能得到2147483648）。
- 如果我们需要更为丰富的数系统，它又会捉襟见肘，比如包含四元数或者和概率相关的数。

糟糕的是，我们根本没有想过自己的需求，就直接轻率的使用Racket中的数作为我们语言中数的表示。

之所以这样做，是因为我们并不关心数本身；我们关心的是诸如将函数作为值这样的编程语言特性。然而，作为语言设计者，你应当在最开始的时候就考虑到这些问题。

接下来讨论闭包的表示。我们其实可以利用Racket的闭包来表示目标语言中的对应概念，与之对应的，用Racket中最基本的函数调用来实现目标语言中的函数调用。

思考题

> 使用Racket函数替换之前闭包的实现。

答案在此：

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value -> Value))])

(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    [numC (n) (numV n)]
    [idC (n) (lookup n env)]
    [appC (f a) (local ([define f-value (interp f env)]
                        [define a-value (interp a env)])
                  ((closV-f f-value) a-value))]
    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) (num* (interp l env) (interp r env))]
    [lamC (a b) (closV (lambda (arg-val)
                         (interp b
                                 (extend-env (bind a arg-val)
                                             env)))))]))
```

练习

> 注意到一个有趣的变化。之前的实现中，环境是在解释appC时被扩展的。这里它是在lamC的解释过程中被扩展的。是这两个中有一个出错了吗？如果不是的话，为什么会出现这种情况？

这种实现方式显然更为简洁，但是我们失去了一项重要的东西：理解。告诉别人源语言中的函数对应于lambda等于什么都没说：如果我们已经知道lambda是干嘛的我们可能就不会花时间去研究它；如果不知道的话，这种直接映射的实现方式也不会教给我们啥（而且很可能会让本来就对该概念一无所知的我们更加困惑）。出于同样的理由，我们没有使用Racket中的状态去理解各种对状态的操作。

然而，一旦我们理解了某个特性，使用它来表示将不再是问题。实际上，这样做会使得我们的解释器更为简洁，毕竟我们不再手工实现所有事情。事实上，如果不使用这种表示方式，后面的一些解释器会变得毫无可读性。【注释】尽管如此，我们还是应该注意防范过度使用宿主语言的特性可能招致的风险。

> 有点像是，“现在我们已经能够通过加一来理解加法，我们可以用加法来定义乘法：不再需要使用加一来定义乘法。”

12.2 错误

当程序出错时，程序员需要得到相应的错误信息。直接使用宿主语言特性可能导致用户收到宿主语言中抛出的错误，这些错误将无法被理解。因此，我们需要谨慎的将各种情况的错误翻译成我们语言的用户所能理解的术语，且不让宿主语言中的错误信息“泄漏过来”。

更糟糕的情形是，那些本应出错的程序可能不会报错！例如，假设我们设计时决定让函数只出现在顶层位置，如果我们没有特意地检测这点，其被去语法糖后得到lambda，最后可能在解释器中被解释得到结果，而它本来应该使解释器出错停止。因此，我们应该极其注意，仅

允许符合期望的表层语言被映射到宿主语言中。

再举个例子，考虑不同的赋值操作。在我们的语言中，给未绑定的变量赋值会导致错误。但是在有些语言中，这种操作会导致该变量被定义。语言设计者常犯的错误是没有很好的确定想要的语义，然后推脱说“它就是实现出来的那个样子”。这种态度（a）是懒惰、马虎的，（b）可能招致不可预料、负面的后果，（c）它使得将语言从一个实现平台移到另一个实现平台变得困难。不要犯这个错误！

12.3 改变含义

将作为值的函数映射为lambda之所以可行是因为我们本来就希望它们拥有相同的含义。但是这种实现方式使得改变函数的含义变得极为困难。让我给你设想一个情形：假设我们想要实现动态作用域。【注释】在我们原来的解释器中，这很简单（历史告诉我们，简直太简单了）。试着在使用了lambda的解释器中实现动态作用域。同样的，将及早求值（eager evaluation）特性映射到惰性求值（lazy application）的语言中（译注，第17章）也是挺有难度的，或者说至少不太容易。

> 只是假设而已。

练习

> 将上面的解释器改成动态作用域的。

重点是，使用自己构造的数据结构并不会使事情更为简单，但一般来说也不会使事情变得更为复杂；与之相对，映射成语言本身特性的方式会使某些特性——通常是宿主语言中已有的特性——的实现极为简单，但是使其他特性的实现变得微妙或困难。还有一个风险是，我们可能并不十分清楚宿主语言的某个特性具体实现了些什么（比如，“lambda”是否真的实现了静态作用域？）。

教训是，仅当我们想要“保留”底层语言的意义时，这才是好用的——甚至是特别明智的，因为它确保我们不会意外地改变其意义。但是，如果我们要利用基础语言的重要组成部分，而只是扩展它的含义，那么其他的实现策略可能也不错（译注，第13章），而不是编写解释器。

12.4 另一个例子

我们再考虑改变一个特性的表示方式。还记得环境是什么吗？

环境是名字到值（如果有赋值的话，那么是名字到地址）的映射。我们通过自建的数据结构实现了这种映射，但是我们可以通过其他方式实现映射吗？当然可以，使用函数就行！这样，环境就变成了读入名字为参数、返回其绑定值（或者报错）的函数：

```
(define-type-alias Env (symbol -> Value))
```

空的环境是什么？对于任何名字的查询都抛出错误的函数：

```
(define (mt-env [name : symbol])
  (error 'lookup "name not found"))
```

（原则上我们应该给它的返回值添加类型注解，应该是Value，但是在这里没啥意义）。给环境添加新的绑定就是创建新函数，该函数检查该名字是不是正在扩展的那个绑定；如果是，直接放回对应的绑定值，如果不是，往被扩展的环境传就行。

```
(define (extend-env [b : Binding] [e : Env])
  (lambda ([name : symbol]) : Value
    (if (symbol=? name (bind-name b))
        (bind-val b)
        (lookup name e))))
```

最后，怎么再环境中查询某个名称呢？调用该环境即可。

```
(define (lookup [n : symbol] [e : Env]) : Value
  (e n))
```

大功告成！

13 语言中支持去语法糖

关于去语法糖（desugaring），之前很多讨论都谈到、用到了，但是我们目前的去语法糖机制是薄弱的。实际上我们用两种不同的方式来使用去语法糖。一方面，我们用它来缩小语言：输入是一个大语言，去语法糖后得到其核心。另一方面，我们也用它来扩展语言：给定现有语言，为其添加新的功能。这表明，去语法糖是非常有用的功能。它是如此之有用，我们该思考一下如下两个问题：

- 我们创建语言的目的是简化常见任务的创建，那么，设计一种支持去语法糖的语言，它会长什么样子呢？请注意，这里的“样子”不仅仅指语法，也包括语言的行为特性。
- 通用语言常常被用作去语法糖的目标，那为什么他们不内建去语法糖的能力呢？比如说，扩展某个基本语言，添加上一个问题的答案所描述的语言。

本章我们将通过研究Racket提供的解决方案同时探索这两个问题。

13.1 第一个例子

> DrRacket有个非常有用的工具叫做Macro Stepper（宏步进器），它能逐步逐步地显示程序的展开。你应该对本章中的所有例子尝试Macro Stepper。不过现在，你应该用#lang plai而不是#lang plai-typed来运行。

回忆一下，前文我们添加`let`时，是将其当作`lambda`的语法糖的。它的模式是：

```
(let (var val) body)
```

被转换为

```
((lambda (var) body) val)
```

思考题

> 如果这听起来不太熟悉，那么现在是时候回忆一下它是怎么运作的了。

描述这个转换最简单的方法就是直接把它写出来，比如：

```
(let (var val) body)
->
((lambda (var) body) val)
```

事实上，这差不多正是Racket语法允许你做的。

> 我们将其命名为`my-let`而不是`let`，因为后者在Racket中已经有定义了。

```
(define-syntax my-let-1  ;定义语法
  (syntax-rules ()       ;语法规则
    [(my-let-1 (var val) body)
     ((lambda (var) body) val)]))
```

`syntax-rules`告诉Racket，只要看到的某个表达式在左括号之后跟的是`my-let-1`，就应该检查它是否遵循模式`(my-let-1 (var val) body)`。这里`var`，`val`和`body`是语法变量：它们是代表代码的变量，可以匹配该位置的任意表达式。如果表达式和模式匹配，那么语法变量就绑定为对应的表达式，并且在右边（的表达式中）可用。

> 您可能已经注意到一些额外的语法，如`()`。我们稍后再解释。

右边（的表达式）——在这里是`((lambda (var) body) val)`——就是最后的输出。每个语法变量都被替换（注意我们的老朋友，替换）其对应的输入部分。这个替换过程非常简单，不会做过多的处理。因此，如果我们尝试这么用

```
(my-let-1 (3 4) 5)
```

第一步Racket不会抱怨3出现在标识符的位置；相反，它会照常处理，去语法糖得

```
((lambda (3) 5) 4)
```

下一步会产生错误：

```
lambda: expected either <id> or `[<id> : <type>]'
  for function argument in: 3
```

这就表明，去语法糖的过程在其功能上直截了当：它不会尝试猜测啥或者做啥聪明事，就是简单的替换重写而已。其输出是表达式，这个表达式也可以被进一步去语法糖。

前文中提到过，这种简单的表达式重写通常使用术语宏（macro）称呼。传统上，这种类型的去语法糖被称为宏展开（macro expansion），不过这个术语有误导性，因为去语法糖后的输出可以比输入更小（通常还是更大啦）。

当然，在Racket中，`let`可以绑定多个标识符，而不仅仅是一个。非正式的写下这种语法的描述的话，比如在黑板上，我们可能会这样写，`(let ([var val] ...) body) -> ((lambda (var ...) body) val ...)`，其中`...`表示“零或更多个”，意思是，输出中的`var ...`要对应输入中的多个`var`。同样，描述它的Racekt语法长的差不多就是这样：

```
(define-syntax my-let-2
  (syntax-rules ()
    [(my-let-2 ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

请注意`...`符号的能力：输入中“对”的序列在输出中变成序列对了；换句话说，Racket将输入序列“解开”了。与之相对，同样的符号也可以用来组合序列。

13.2 用函数实现语法变换器

之前我们看到，my-let-1并不会试图确保标识符位置中的语法是真正的（即语法上的）标识符。用syntax-rules机制我们没法弥补这一点，不过使用更强大的机制，称为syntax-case，就可以做到。由于syntax-case还有很多其他有用的功能，我们分步来介绍它。

首先要理解的是，宏实际上是一种函数。但是，它并不是从常见的运行时值到（其他）运行时值的函数，而是从语法到语法的函数。这种函数执行的目的是创建要被执行的程序。注意这里我们说的是要被执行的程序：程序的实际执行可能会晚得多（甚至根本不执行）。看看去语法糖的过程，这点就很清楚了，很显然它是（一种）语法到（另一种）语法的函数。两个方面可能导致混淆：

- `syntax-rules`的表示中并没有明确的参数名或者函数头部，可能没有明确表明这是一个转换函数（不过重写规则的格式有暗示这个事实）。
- 去语法糖指的是，有个（完整的）函数完成了整个过程。这里，我们实际写的是一系列小函数，每个函数处理一种新的语法结构（比如my-let-1），这些小函数被某个看不见的函数组合起来，完成整个重写过程。（比如说，我们并没有说明，某个宏展开后的输出是否还会进一步被展开——不过简单试一下就知道，事实确实如此。）

练习

> 编写一个或多个宏，以确定宏的输出会被进一步展开。

还有个微妙之处。宏的外观和Racket代码非常类似，并没有指明它“生活在另一个世界”。想象宏定义使用的是完全不同的语言——这种语言只处理语法——写就很有助于我们建立抽象。然而，这种简化并不成立。现实中，程序变换器——也被称为编译器（compiler）——也是完整的程序，它们也需要普通程序所需要的全部功能。也就是说我们还需要创立一种平行语言，专门处理程序。这是浪费和毫无意义的；因此，Racket自身就支持语法转换所需的全部功能。

背景说完了，接下来开始介绍`syntax-case`。首先我们用它重写my-let-1（重写时使用名字my-let-3）。第一步还是先写定义的头部；注意到参数被明确写出：

```
<sc-macro-eg> ::=  ;syntax-case宏，示例

    (define-syntax (my-let-3 x)
      <sc-macro-eg-body>)
```

`x`被绑定到整个`(my-let-3 ...)`表达式

你可能想到了，`define-syntax`只是告诉Racket你要定义新的宏。它不会指定你想要实现的方式，你可以自由地使用任何方便的机制。之前我们用了`syntax-rules`；现在我们要用`syntax-case`。对于`syntax-case`，它需要显式的被告知要进行模式匹配的表达式：

```
<sc-macro-eg-body> ::=

    (syntax-case x ()
      <sc-macro-eg-rule>)
```

现在可以写我们想要表达的重写规则了。之前的重写规则有两个部分：输入结构和对应的输出。这里也一样。前者（输入匹配）和以前一样，但后者（输出）略有不同：

```
<sc-macro-eg-rule> ::=

    [(my-let-3 (var val) body)
     #'((lambda (var) body) val)]
```

关键是多出了几个字符：`#'`。让我们来看看这是什么。

在`syntax-rules`中，输出部分就指定输出的结构。与之不同，`syntax-case`揭示了转换过程函数的本质，因此其输出部分实际上是任意表达式，该表达式可以执行任何它想要进行的计算。该表达式的求值结果应该是语法。

语法其实是个数据类型。和其他数据类型一样，它有自己的构造规则。具体来说，我们通过写`#'`来构造语法值；之后的那个s-expression被当作语法值。（顺便提一句，上面宏定义中的`x`绑定的也是这种数据类型。）

语法构造器`#'`有种特殊属性。在宏的输出部分中，所有输入中出现的语法变量都被自动绑定并替换。因此，比方说，当展开函数在输出中遇到var时，它会将var替换为相应的输入表达式。

思考题

> 在上述宏定义中去掉`#'`试试看。后果如何？

到目前为止，syntax-case似乎只是更为复杂的syntax-rules：唯一稍微好些的地方是，它更清楚地描述了展开过程的函数本质，同时明确了输出的类型，但其他方面则更加笨拙。但是，我们将会看到，它还提供了强大的功能。

练习

> 事实上，syntax-rules可以被表述为基于syntax-case的宏。请定义这个宏。

13.3 防护装置

现在我们可以回过来考虑到最初引致syntax-case的问题：确保my-let-3的绑定位置在语法上是标识符。为此，您需要知道syntax-case的一个新特性：每一条重写规则可以包含两个部分（如同前面的例子），也可以包含三个部分。如果有三个部分，中间那个被视为防护装置（guard）:它是一个判断，仅当其计算值为真时，展开才会进行，否则就报告语法错误。在这个例子中，有用的判断函数是`identifier?`，它能判定某个语法对象是否是标识符（即变量）。

思考题

> 写出防护装置，并写出包含防护（装置）的（重写）规则。

希望你发现了其中的微妙之处：`identifier?`的参数是语法类型的。要传给它的是绑定到var的实际语法片段。回想一下，var是在语法空间中绑定的，而`#'`会替换其中的绑定变量。因此，这里防护装置的正确写法是：

```
(identifier? #'var)
```

有了这些信息，我们现在可以写出整个规则：

```
<sc-macro-eg-guarded-rule> ::=

    [(my-let-3 (var val) body)
     (identifier? #'var)
     #'((lambda (var) body) val)]
```

思考题

> 现在有了带防护的规则定义，尝试使用宏，在绑定位置使用非标识符，看看会发生什么。

13.4 Or：简单但是包含很多特性的宏

考虑`or`，它实现或操作。使用前缀语法的话，自然的做法是允许`or`有任意数目的子项。我们把`or`展开为嵌套的条件（表达式），以此判断表达式的真假。

13.4.1 第一次尝试

试试这样的or：

```
(define-syntax (my-or-1 x)
  (syntax-case x ()
    [(my-or-1 e0 e1 ...)
     #'(if e0
           e0
           (my-or-1 e1 ...))]))
```

它说，我们可以提供任何数量的子项（待会儿再解释这点）。（宏）展开将其重写为条件表达式，其中的条件是第一个子项；如果该项为真值，就返回这个值（待会再讨论这点！），否则就返回其余项的或。

我们来试一个简单的例子。这应该计算为真，但是：

```
> (my-or-1 #f #t)
my-or-1: bad syntax in: (my-or-1)
```

发生了什么？这个表达式变成了

```
(if #f
    #f
    (my-or-1 #t))
```

继续展开

```
(if #f
    #f
    (if #t
        #t
        (my-or-1)))
```

对此我们没有定义。这是因为，模式`e0 e1 ...`表示一个或更多子项，但是我们忽略了没有子项的情况。

没有子项时应该怎么办？或运算的单位元是假值。

练习

> 为什么正确的默认值是`#f`？

我们可以通过加上这条规则，展示不止一条规则的宏。宏的规则是顺序匹配的，所以我们必须把最具体的规则放在最前面，以免它们被更一般的规则覆盖（尽管在这个例子中，两条规则并不重叠）。改进后的宏是：

```
(define-syntax (my-or-2 x)
  (syntax-case x ()
    [(my-or-2)
     #'#f]
    [(my-or-2 e0 e1 ...)
     #'(if e0
           e0
           (my-or-2 e1 ...))]))
```

现在宏可以和预期一样展开了。虽然没有必要，但是我们加上一条规则，处理只有一个子项的情况：

```
(define-syntax (my-or-3 x)
  (syntax-case x ()
    [(my-or-3)
     #'#f]
    [(my-or-3 e)
     #'e]
    [(my-or-3 e0 e1 ...)
     #'(if e0
           e0
           (my-or-3 e1 ...))]))
```

这使展开的输出更加简约，对后文中我们的讨论是有帮助的。

> 注意到在这个版本的宏中，规则不再是互不重叠的了：第三条规则（一个或多个子项）包含了第二条（一个子项）。因此，第二条规则与第三条不能互换，这是至关重要的。

13.4.2 防护装置的求值

之前说这个宏的展开符合我们的预期，是吧？试试这个例子：

```
(let ([init #f])
  (my-or-3 (begin (set! init (not init))
                  init)
           #f))
```

请注意，or返回的是第一个“真值”的值，以便程序员在进一步的计算中使用它。因此，这个例子返回init的值。我们期望它是什么？因为我们已经翻转了init的价值，自然而然的，我们期望它返回 `#t` 。但是计算得到的是 `#f` ！

> 这里的问题不在 `set!` 。比如说，如果我们在这里不放赋值，而是放上打印输出，那么打印输出就会发生两次。

要理解为何如此，我们必须检查展开后的代码：

```
(let ([init #f])
  (if (begin (set! init (not init))
             init)
      (begin (set! init (not init))
             init)
      #f))
```

啊哈！因为我们把输出模式写成了

```
#'(if e0
      e0
      ...)
```

当我们第一次写下它时，看起来完全没有问题，而这正表明了编写宏（或，其他的程序转换系统）时的一个非常重要的原则：不要复制代码！在我们的设定中，语法变量永远不应被重复；如果你需要重复某个语法变量，以至于它所代表的代码会被多次执行，请确保已经考虑

到了这么做的后果。或者，如果只需要该表达式的值，那么绑定一下，接下来使用绑定标识符的名字就好。示例如下：

```
(define-syntax (my-or-4 x)
  (syntax-case x ()
    [(my-or-4)
     #'#f]
    [(my-or-4 e)
     #'e]
    [(my-or-4 e0 e1 ...)
     #'(let ([v e0])
         (if v
             v
             (my-or-4 e1 ...)))]))
```

这个引入绑定的模式会导致潜在的新问题：你可能会对不必要的表达式求值。事实上，它还会导致第二个、更微妙的问题：即使该表达式需要被求值，你可能在错误的上下文中对其求值了！因此，你必须仔细推敲表达式是否要被求值，如果是的话，只在正确的地方求一次值，然后存贮其值以供后续使用。

用 `my-or-4` 重复之前包含 `set!` 的例子，结果是 `#t` ，符合我们的预期。

13.4.3 卫生

希望你现在觉得没啥问题了。

思考题

> 还有啥问题？

考虑这个宏 `(let ([v #t]) (my-or-4 #f v))` 。我们希望其计算的结果是啥？显然是 `#t` ：第一个分支是 `#f` ，但第二个分支是 `v` ， `v` 绑定到 `#t` 。但是观察展开后：

```
(let ([v #t])
  (let ([v #f])
    (if v
        v
        v)))
```

直接运行该表达式，结果为 `#f` 。但是， `(let ([v #t]) (my-or-4 #f v))` 求值得 `#t` 。换种说法，这个宏似乎神奇地得到了正确的值：在宏中使用的标识符名称似乎与宏引入的标识符无关！当它发生在函数中时，并不令人惊讶；宏展开过程也享有这种特性，它被称为卫生（hygiene）。

理解卫生的一种方法是，它相当于自动将所有绑定标识符改名。也就是说，程序的展开如下：

```
(let ([v #t])
  (or #f v))
```

变成

```
(let ([v1 #t])
  (or #f v1))
```

（注意到v一致的重命名为v1），接下来变成

```
(let ([v1 #t])
  (let ([v #f])
       v
       v1))
```

重命名后变成

```
(let ([v1 #t])
  (let ([v2 #f])
       v2
       v1))
```

此时展开结束。注意上述每一个程序，如果直接运行的话，都会产生正确的结果。

13.5 标识符捕获

卫生宏解决了语法糖的创造者常常会面对的重要痛点。然而，在少数情况下，开发人员需要故意违反卫生原则。回过来考虑对象，对于这个输入程序：

```
(define os-1
  (object/self-1
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

（对应的）宏应该是什么样的？试试这样：

```
(define-syntax object/self-1
  (syntax-rules ()
    [(object [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                   ...)))
         self))]))
```

不幸的是，这个宏会产生以下错误：

```
self: unbound identifier in module in: self
;self: 未绑定的标识符
```

错误指向的是first方法体中的self。

练习

> 给出卫生展开的步骤，理解为何报错是我们预期的结果。

在正面解决该问题之前，让我们考虑输入项的一种变体，使绑定显式化：

```
(define os-2
  (object/self-2 self
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

对应的宏只需要稍加修改：

```
(define-syntax object/self-2
  (syntax-rules ()
    [(object self [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                 ...)))
         self))]))
```

这个宏展开正确。

习题

> 给出这个版本的展开步骤，看看不同在哪里。

洞察其中的区别：如果进入绑定位置的标识符是由宏的用户提供的话，那么就没有问题了。因此，我们想要假装引入的标识符是由用户编写的。函数 `datum->syntax` 接收两个参数，第一个参数是语法，它将第二个参数——s-expression——转换为语法，假装其是第一个参数的一部分（在我们的例子中，就是宏的原始形式，它被绑定为x）。为了将其结果引入到用于展开的环境中，我们使用 `with-syntax` 在环境中进行绑定：

```
(define-syntax (object/self-3 x)
  (syntax-case x ()
    [(object [mtd-name (var) val] ...)
     (with-syntax ([self (datum->syntax x 'self)])
       #'(let ([self (lambda (msg-name)
                       (lambda (v) (error 'object "nothing here")))])
           (begin
             (set! self
                   (lambda (msg-name)
                     (case msg-name
                       [(mtd-name) (lambda (var) val)]
                       ...)))
             self)))]))
```

于是我们可以隐式的使用self了：

```
(define os-3
  (object/self-3
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

13.6 对编译器设计的影响

在一个语言的定义中使用宏对所有其工具都有影响，特别是编译器。作为例子，考虑`let`。`let`的优点是，它可以被高效的编译，只需要扩展当前环境就行了。相比之下，将`let`展开成函数调用会导致更昂贵的操作：创建闭包，再将其应用于参数，实际上获得的效果是一样的，但是花费更多时间（通常还要更多空间）。

这似乎是反对使用宏的论据。不过，聪明的编译器会发现这个模式老是出现，并会在其内部将左括号左括号lambda转换回`let`的等价形式。这么做有两个好处。第一个好处是，语言设计者可以自由地使用宏来获得更小的核心语言，而不必与执行成本进行权衡。

第二个好处更微妙。因为编译器能识别这个模式，其他的宏也可以利用它并获得相同的优化；它们不再需要扭曲自己的输出，如果自然的输出恰好是左括号左括号lambda，将其再转化成let（否则就必须这么做）。比如说，在编写某些模式匹配（的宏）的时候，左括号左括号lambda模式就会自然的出现，而想要将其转换为let的话就必须多做一步——现在不必要了。

13.7 其他语言中的去语法糖

不仅仅是Racket，许多现代语言也通过去语法糖来定义操作。例如在Python中，for迭代就是语法模式。程序员写下`for x in o`时，他

- 引入了新标识符（称之为i，但是，不要让其捕获了程序员定义的i，即，卫生的绑定i！），
- 将其绑定到从o获得的迭代器（iterator），
- 创建（可能）无限的while循环，反复调用i的.next方法，直到迭代器引发StopIteration异常。

现代编程语言中有许多这样的模式。

14 控制指令

术语控制指的是编程语言中任何使得计算过程前进的指令，因为它“控制”了计算机的程序计数器（program counter）。从这个意义上说，即使是简单的算术表达式也应该被认为是一种“控制”，而像顺序执行、函数调用和返回这样的操作，就更应该是了。不过，实践中我们通常用这个名词指代那些导致控制非局部转移的——尤其是除了函数、过程以及将要学到的异常（exception）之外的——指令。本章我们将学习这类指令。

在研究这些控制指令时，需要指出的是，即使没有它们，我们的语言也是图灵完备的，也就是说我们并没有获得额外的“能力”。因此，控制指令所做的是，改变、改善我们的表达方式，从而增强程序的结构。所以，专注于程序的结构有益于本章的学习。

14.1 Web上的控制

让我们从研究Web程序的结构开始。考虑下面的程序：【注释】

```
(display
  (+ (read-number "First number")
     (read-number "Second number")))
```

> 今后，我们将把它称为“加法服务”。当然，你应该将它理解为更为复杂应用的一个简化版。例如，应用可能提示输入的是旅程的起点和目的地，加法对应的实际服务可能是根据输入的起点终点计算航线或者机票的价格。在两个（输入）步骤之间甚至可能也有计算：例如，在输入第一个城市后，航空公司可能会提示我们可供选择的目的地。

为了测试这些想法，下面是read-number的实现：

```
(define (read-number [prompt : string]) : number
  (begin
    (display prompt)
    (let ([v (read)])
      (if (s-exp-number? v)
          (s-exp->number v)
          (read-number prompt)))))
```

在控制台或DrRacket中运行时，该程序会提示我们输入一个数字，然后输入另一个数字，最后显示它们的总和。

现在假设我们想在Web服务器上运行。我们立即遇到难点：服务器端Web程序的结构是这样的：它们生成一个网页，比如请求第一个数字的网页，然后停止。结果，程序的其余部分——在这里，提示第二个数字，然后求和，然后打印结果——丢失了。

思考题

> 为什么Web服务器的行为如此奇怪？

这种行为至少有两个原因：一个也许是历史的，另一个是技术的。历史原因是Web服务器最初设计为供应页面，即静态内容。任何程序的运行都必须将其输出生成为文件，服务器将该文件提供给客户端。很自然的，开发人员想到为什么同样的程序在web上就不能按需运行。于是，后来Web上出现了动态内容。构成Web应用的最小增量单元不再是页面，而是一个个执行结束后生成页面各个部分所需内容的程序。

更重要的原因——也是导致目前状况的原因——是技术性的。想象一下，我们的加法服务器已经生成了第一个提示。回想一下，有相当多的计算要进行：第二个提示，求和和显示结果。这些计算必须暂停，等待用户的输入。如果有成千上万的用户，那么必须暂停成千上万的计算，这会产生巨大的性能问题。此外，假设用户实际上没有完成计算——类似于在网上书店或航空公司网站上搜索，而不完成购买。服务器如何知道何时终止计算，甚至是否终止计算？而在终止之前，与该计算相关的资源仍被占用。

因此，Web协议从其概念上就被设计为无状态的（stateless）：它不将与中间计算相关的状态存储在服务器上。这使得Web程序员被迫在其他地方维护所有必要的状态，每个请求都需要携带能够完全恢复计算所需的状态。在实践中，Web并不都是完全无状态的，但是它们在很大程度上倾向这个方向，因此研究这类程序的结构是非常有教益的。

接下来考虑一下客户端的Web程序：那些在浏览器中运行的程序，通常用JavaScript编写，或被编译成JavaScript。假设某个计算需要与服务器进行通信。（JavaScript提供的）指令为XMLHttpRequest。用户创建这个指令的实例，然后调用其 `send` 方法向服务器发送消息。然而，与服务器通信并不是即时的（并且根据网络的状态，实际上可能永远不会完成）。这导致发送进程被挂起。

JavaScript的设计者决定让该语言是单线程的，即，任意时间只能有一个线程在执行。【注释】这避免了赋值与线程结合而产生的各种风险。因此，JavaScript进程会被锁定以等待响应，这期间不可能做任何其他事情：例如，页面上的其他处理程序不再响应。

> 因为这会导致结构性问题，现在有各种提议，实际上是要为JavaScript添加“安全的”线程。本章所描述的想法可以被看作是另一种方案，提供类似的结构优势。

为了避免这个问题，XMLHttpRequest的设计要求开发者提供一个函数来响应请求（请求到达时将调用该程序）。该回调函数在系统中注册。需要传递请求结果给该回调函数让其完成后续处理过程。因此，并非处于性能方面的考虑，而是为了避免同步、非原子性和死锁问题，客户端Web也发展出相同的程序模式。让我们更好地理解这种模式。

14.1.1 将程序分解成现在和以后

我们来考虑如何让上述程序在无状态的环境下——比如在Web服务器上——工作。首先我们需要确定第一个交互，是提示输入第一个数字，因为Racket从左到右计算参数。将程序分成两部分是有益的：第一个交互产生啥（现在就可以运行），以及之后需要发生什么（必须以某种方式“记住”）。前者很容易：

```
(read-number "First number")
```

我们已经用文字解释过剩下的东西了，但是现在是时候把它写成程序了。似乎应该类似于【注释】

```
(display
  (+ <第一个交互的返回值>
     (read-number "Second number")))
```

> 我们现在故意忽略read-number部分，但会回过来讨论它。现在，我们假设它是内置的。

但是，Web服务器不能执行这个东西，因为它显然不是程序。我们需要一种方式将其写成程序。

观察一下这个计算的特点：

- 它是合法的程序。
- 它需要保持暂停状态，直到请求进入。
- 它需要某种方式——例如参数——来引用前一个交互的值。

综合这些特点，显然我们应该将其表示为函数：

```
(lambda (v1)
  (display
    (+ v1
       (read-number "Second number"))))
```

14.1.2 部分的解决方案

在Web上，还有个额外的问题：每个带有输入元素的Web页面都需要引用存储在Web上的程序，该程序将从表单接收数据并对其进行处理。这个程序是在表单的action字段中指明的。因此，设想服务器生成一个新的标签，将前述函数存储在与该标签相关联的表格中，并且在action字段中引用该表格。如果客户端最终提交了表单，这个时候，服务器提取出关联的函数，向其提供表单的值，从而恢复执行。

思考题

> 上述方案是无状态的吗？

假设我们在自定义的Web服务器上维护这么一个表格。在这个服务器上，可能会有一个特殊版本的read-number，称之为call-read-number/suspend，记录程序的其余部分：

```
(read-number/suspend "First number"
                     (lambda (v1)
                       (display
                        (+ v1
                           (read-number "Second number")))))
```

为了测试，我们来实现这个子程序。首先，我们需要标签的表示法；用数字就好：

```
(define-type-alias label number)
```

假设`new-label`在每次调用时都会生成新标签。

练习

> 定义`new-label`。需要的话参考`new-loc`以获得灵感。

需要一个表，来存储代表程序其余部分的子程序。

```
(define table (make-hash empty))
```

存储这些子程序：

```
(define (read-number/suspend [prompt : string] rest)
  (let ([g (new-label)])
    (begin
      (hash-set! table g rest)
      (display prompt)
      (display " To enter it, use the action field label ")
      (display g))))
```

现在运行上面的read-number/suspend调用，系统会打印

```
First number To enter it, use the action field label 1
```

这就相当于，在Web页面中打印提示，并在action字段中放入“标签1”。因为我们在模拟网页，需要有个东西来表示浏览器的提交过程。这里需要标签（来自action字段）和表单中输入的值。给定了这两个值，这个子程序需要从表中提取出相关子程序，并将其应用于表单值。

```
(define (resume [g : label] [n : number])
  ((some-v (hash-ref table g)) n))
```

有了这些，我们现在可以模拟输入3并点击“提交”按钮的行为，运行：

```
> (resume 1 3)
```

其中1是标签，3是用户输入。不幸的是，这么做只会产生另一个提示，因为我们还没有完成程序的转换。要去除read-number，我们需要转换整个程序：

```
(read-number/suspend "First number"
                     (lambda (v1)
                       (read-number/suspend "Second number"
                                            (lambda (v2)
                                              (display
                                               (+ v1 v2))))))
```

为了安全起见，我们还可以在read-number/suspend结束的地方添加报错，从而确保计算在每次输出之后终止（以确保“挂起”的最极端形式）。

执行这个程序时，必须两次使用resume：

```
First number To enter it, use the action field label 1
halting: Program shut down
> (resume 1 3)
Second number To enter it, use the action field label 2
halting: Program shut down
> (resume 2 10)
13
```

其中两次用户输入分别是3和10，总和给出是13，而

```
halting
```

信息是我们添加的报错命令生成的。

我们故意略去了程序中某些有趣部分的类型。来看看这些类型应该是什么。read-number/suspend的第二个参数是读入数字并返回最终结果的子程序：`(number -> 'a)`。同样，resume的返回类型也是`'a`。这些`'a`如何相互沟通？是通过将标签映射到`(number -> ’a)`的表完成的。也就是说，计算过程中的每一步都产生相同类型的结果。`read-number/suspend`写入表中，`resume`从表中读取。

14.1.3 实现无状态

实际上我们并没有实现无状态，因为服务器上有一大张表，而我们缺乏明确手段去除此表。如果可以完全避免服务器上的状态就好了。这意味着我们必须将相关的状态移交给客户端。

服务器实际上以两种方式持有了状态。其一，可以存放任意多个——而不是常数个（比如线性相关于程序本身的大小）——条目的哈希表，。其二，我们在表中存放的是实实在在的闭包，而闭包中可以保有任意数量的状态。我们很快就会更清楚地看到这一点。

先从消除闭包开始着手。我们可以把所有的函数参数改成实名的全局函数（这迫使我们只会拥有有限个闭包，因为程序的长度不可能是无限的）：

```
(read-number/stateless "First number" prog1)

(define (prog1 v1)
  (read-number/stateless "Second number" prog2))

(define (prog2 v2)
  (display (+ v1 v2)))
```

注意到每块代码都只引用下一块代码的名称，而没有引入真正的闭包。参数的值来自于表单。唯一的问题是：prog2中的v1是未绑定的标识符！

解决这个问题的方法是，不要在每一步之后创建闭包，而是将v1发送到客户端并存储在那里。存储在哪里呢？浏览器为此提供了两种机制：**Cookie**和隐藏字段。我们用哪一个？

14.1.4 与状态互动

Cookie和隐藏字段之间的本质区别是，所有页面共享相同的**cookie**，但每个页面都包含自己的隐藏字段。

先来考虑与现有程序的一串交互，（在两个地方都）使用read-number/suspend。就像这样：

```
First number To enter it, use the action field label 1
> (resume 1 3)
Second number To enter it, use the action field label 2
> (resume 2 10)
13
```

因此，恢复标签2似乎表示将3加到给定的参数（即，表单字段值）。保险起见，

```
> (resume 2 15)
18
```

一切正常。现在假设我们再次使用标签1：

```
> (resume 1 5)
Second number To enter it, use the action field label 3
```

注意，需要使用标签3，而不是标签1来恢复这个新的程序执行。的确，

```
> (resume 3 10)
15
```

但是我们应该问，如果重用标签2会发生什么？

思考题

> 试试 `(resume 2 10)` 。

这就是恢复之前的计算。因此，我们期望它产生和之前一样的结果：

```
> (resume 2 10)
13
```

现在来创建一个有状态的实现。通过共享一个可变状态但是拥有自己环境的闭包可以模拟这种行为。所以我们可以这样做，使用现有的read-number/suspend，但是不依赖lambda的闭包行为，即不使用任何自由变量。

```
(define cookie '-100)

(read-number/suspend "First number"
                     (lambda (v1)
                       (begin
                         (set! cookie v1)
                         (read-number/suspend "Second number"
                                              (lambda (v2)
                                                (display
                                                 (+ cookie v2)))))))
```

练习

> 对于之前的交互序列，现在的期望值是啥？

思考题

> 计算过程是什么样的？

起初，似乎没啥不同：

```
First number To enter it, use the action field label 1
> (resume 1 3)
Second number To enter it, use the action field label 2
> (resume 2 10)
13
```

当再次使用最初的计算时，我们确实得到新的恢复标签：

```
> (resume 1 5)
Second number To enter it, use the action field label 3
```

使用新标签时，计算结果如我们所期望的：

```
> (resume 3 10)
15
```

关键的一步来了：

```
> (resume 2 10)
15
```

标签2的两次恢复产生了不同的答案，这一点不足为奇，因为它们依赖于可变状态。问题是，当我们将相同的行为转换到Web时会发生什么。

想象一下，访问某旅馆预订网站，寻找某个城市的旅馆。返回的网页中，你看到一个旅馆的链表和标签1。你在新（浏览器）标签或窗口中浏览其中的一个旅馆；这个页面中生成了那个旅馆的信息，还有标签2用作预订旅馆。然而，你返回旅馆链表，并在新的标签或窗口中查看了另一家旅馆。这产生了第二家旅馆的信息，还有标签3用作该旅馆的预订。然而，你决定选择第一家旅馆，返回第一家旅馆的页面，然后选择预订按钮，也就是提交了标签2。你想要预订的是哪家旅馆？尽管你预期订的是第一家，大多数旅游网站上，你要么预订了第二家旅馆——即最后查看的，而不是预订按钮所在的网页上的那家——要么被报告错误。这是因为在Web站点普遍使用了cookie，这是大多数Web API所鼓励的做法。

14.2 Continuation传递模式

之前所说的函数是有名称的。虽然用Web描述问题，但是我们用的是更古老的概念：这类函数被称为**continuation**（延续），而这种风格的程序被称为**continuation-passing style**（Continuation传递模式，简称CPS）。【注释】这值得研究一下，因为它是学习其他各种非平凡控制指令——如生成器——的基础。

> 我们会自由地将CPS当作名词和动词使用：一种特定的代码模式，将代码转化为此种模式。

此前，我们将程序转化为，没有Web输入操作嵌套在另一个中。动机很简单：当程序终止时，所有嵌套的计算都会丢失。对于XMLHttpRequest来说，类似的论据（在程序本地意义上）成立：所有依赖于Web服务器响应结果的计算，都需要驻留在对服务器请求相关联的回调中。

事实上，我们并不需要转化每一个表达式。只需要处理涉及实际Web交互的表达式。比如说，如果要进行的计算不是加法，而是比它复杂得多的数学表达式，这个数学表达式我们是不需要转换的（不涉及Web交互）。不过，如果这里有个函数调用，那么我们必须绝对确定这个函数、它调用的函数、这些函数调用的函数（整个调用链）中不存在任何的Web调用，才可以不对它进行转换。否则，保险起见，我们必须转化所有的这些函数。总之，我们必须转化每个我们无法确定不执行任何Web交互的表达方式。

因此，这里转化的核心就是把每个单参数函数 `f` 转换成具有额外参数的函数。这个额外的参数就是continuation，代表了其余的计算。Continuation本身也是单参数的函数。这个参数的输入是 `f` 本来的返回值，后续计算本来需要使用这个返回值继续。转换后 `f` 将不再返回值，而是将原来的返回值传递给它的continuation。

CPS是种通用的转化，可以作用在任何程序上。因为它是一种程序转换，所以我们可以把它看作是特殊的去语法糖：特别之处是，它不是把程序从大语言转化到小语言（类似于宏），或者从一种语言转化到另一种语言（就像编译器那样），而是在同一种语言中的程序转换：

从完整语言转化到受限制的形式，遵从这里讨论的模式。因此，我们可以使用完整语言的求值器对CPS程序求值。

14.2.1 用去语法糖实现

我们已经对去语法糖有了很好的支持，所以我们来它来定义CPS转换。具体来说，我们将实现CPS宏。为了更加干净地将源语言与目标语言分开，我们所使用的大部分语言结构都会用略有不同的名称：单变量的rec和with而不是let和letrec；lam而不是lambda；cnd而不是if；seq取代begin；set取代set!。这会是足够丰富的语言，可以编写一些有趣的程序！

> 后文中宏的子句按照我认为从容易到困难的顺序排列。但是，宏定义的代码必须避免模式的重复，因此遵循不同的顺序。

```
<cps-macro> ::=  ;CPS宏

    (define-syntax (cps e)
      (syntax-case e (with rec lam cnd seq set quote display read-number)
        <cps-macro-with-case>
        <cps-macro-rec-case>
        <cps-macro-lam-case>
        <cps-macro-cnd-case>
        <cps-macro-display-case>
        <cps-macro-read-number-case>
        <cps-macro-seq-case>
        <cps-macro-set-case>
        <cps-macro-quote-case>
        <cps-macro-app-1-case>
        <cps-macro-app-2-case>
        <cps-macro-atomic-case>))
```

我们的CPS表示法会将每个表达式转变成单参数的函数，参数就是continuation。转换后的表达式最终要么提供值调用continuation，要么将continuation传递给其他表达式，归纳地说，其他表达式也遵从这个不变量关系，因此最终continuation会被提供某个值。所以说，所有的CPS输出看起来都类似于`(lambda (k) ...)`（我们将依赖卫生来保证所有引入的k不会相互冲突）。

首先，我们来处理简单的情况，原子值。尽管概念上来说它是最简单的，但是我们将其放在最后一项，因为放在前面的话它会遮盖掉其他匹配。（理想情况下，我们应该将其放在第一个位置，然后提供一个能精确定义我们原子值的匹配表达式，这里放宽要求是因为我们对其他情况更为关心。）原子值的情况中，我们已经有一个值，将其传递给continutaion即可：

```
<cps-macro-atomic-case> ::=  ;原子

    [(_ atomic)
     #'(lambda (k)
         (k atomic))]
```

被引用的常量也一样处理：

```
<cps-macro-quote-case> ::=

    [(_ 'e)
     #'(lambda (k) (k 'e))]
```

我们还知道，with和rec可以当作宏来处理：

```
<cps-macro-with-case> ::=

    [(_ (with (v e) b))
     #'(cps ((lam (v) b) e))]

<cps-macro-rec-case> ::=

    [(_ (rec (v f) b))
     #'(cps (with (v (lam (arg) (error 'dummy "nothing")))
                  (seq
                   (set v f)
                   b)))]
```

赋值也是容易的：先求出新的值，然后再执行实际的更新操作：

```
<cps-macro-set-case> ::=

    [(_ (set v e))
     #'(lambda (k)
         ((cps e) (lambda (ev)
                    (k (set! v ev)))))]
```

序列指令也是直白的：依次执行每个操作。请注意我们保持了序列的语义：不仅遵守了操作的顺序，第一个子项（e1）的值在第二个（e2）的计算中不会被用到，所以该值所绑定到的标识符的名称也就无关紧要。

```
<cps-macro-seq-case> ::=

    [(_ (seq e1 e2))
     #'(lambda (k)
         ((cps e1) (lambda (_)
                     ((cps e2) k))))]
```

处理条件指令时，需要创建新的continuation，用来记住我们在等待条件表达式的求值结果。不过，一旦获得了其值，根据其值的不同我们可以选择进入已有的continuation分支。

```
<cps-macro-cnd-case> ::=

    [(_ (cnd tst thn els))
     #'(lambda (k)
         ((cps tst) (lambda (tstv)
                      (if tstv
                          ((cps thn) k)
                          ((cps els) k)))))]
```

处理函数调用时，有两种情况需要考虑。我们必须要处理语言中创建的函数，也就是单参数函数。然而，为了编写示例程序，能够使用诸如+和*之类的指令很有用。因此，为了简单起见，我们将假定单参数函数是用户编写的，因此需要CPS转换，而双参数函数是不会执行任何Web或其他控制操作的指令，因此可以直接调用；我们还假定原生指令可以直接写出（即，函数位置不是复杂表达式，本身不会执行Web交互）。

对于函数调用，我们必须先对函数和参数表达式求值，一旦获取了这些就可以实际进行函数的调用。因此我们很容易将函数调用的转换写成这样：

```
<cps-macro-app-1-case-take-1> ::=

    [(_ (f a))
     #'(lambda (k)
         ((cps f) (lambda (fv)
                    ((cps a) (lambda (av)
                               (k (fv av)))))))]
```

思考题

> 你看出为什么这是错的吗？

问题在于，虽然函数现在是值了，也就是闭包，其函数体可以很复杂：比如说，对函数体求值可以导致进一步的Web交互，此时函数体的其余部分，包括待处理的`(k ...)`（即程序的其余部分）将全部丢失。为了避免这种情况，我们必须把k提供给函数的值，让归纳不变量保证k最终会被调用于fv作用于av的得到的值：

```
<cps-macro-app-1-case> ::=

    [(_ (f a))
     #'(lambda (k)
         ((cps f) (lambda (fv)
                    ((cps a) (lambda (av)
                               (fv av k))))))]
```

处理内置双目操作的特殊情况比较容易：

```
<cps-macro-app-2-case> ::=

    [(_ (f a b))
     #'(lambda (k)
         ((cps a) (lambda (av)
                    ((cps b) (lambda (bv)
                               (k (f av bv)))))))]
```

用户定义的函数不能使用这个模式，因为我们假设这里f的调用总是会返回，而不进行任何不寻常的控制转移。

函数本身就是一种值，该值本身应该被返回给挂起的计算（一个continuation）。然而，前面函数调用的情况表明，函数转化后需要传入额外的参数——调用点的continuation。这就留下一个问题：该向函数体提供哪个continuation？

```
<cps-macro-lam-case-take-1> ::=

    [(_ (lam (a) b))
     (identifier? #'a)
     #'(lambda (k)
         (k (lambda (a dyn-k)
              ((cps b) ...))))]
```

也就是说，在这里的...位置上，我们该填入k还是dyn-k？

思考题

> 该填入哪个continuation呢？

前者是闭包创建位置的continuation。后者是闭包调用位置的continuation。换一种说法，前者是“静态的”，后者是“动态的”。这里，我们需要使用动态的continuation，否则会发生非常奇怪的事情：程序会返回到创建闭包的地方，而不是它被使用的地方！这会导致非常奇怪的程序行为，所以我们避免这么做。请注意，这里我们有意识地选择动态的continuation，就如同在处理作用域时，我们选择了静态的环境。

```
<cps-macro-lam-case> ::=

    [(_ (lam (a) b))
     (identifier? #'a)
     #'(lambda (k)
         (k (lambda (a dyn-k)
              ((cps b) dyn-k))))]
```

最后，为了建模Web编程的目的，我们需要添加输入和输出指令。输出遵循前述函数调用的模式：

```
<cps-macro-display-case> ::=

    [(_ (display output))
     #'(lambda (k)
         ((cps output) (lambda (ov)
                         (k (display ov)))))]
```

对于输入，使用现有的read-number/suspend就可以了，不过这里由我们来生成其使用，而不是让程序员来创建：

```
<cps-macro-read-number-case> ::=

    [(_ (read-number prompt))
     #'(lambda (k)
         ((cps prompt) (lambda (pv)
                         (read-number/suspend pv k))))]
```

请注意，绑定为k的continuation就是在Web交互处我们需要存储的continuation。

测试CPS转换后的代码有些小麻烦，因为所有CPS项都需要读入continuation。最初的continuation可以是（a）读入值并返回它，或者（b）读入值并打印它，或者（c）读入值，打印它并准备好进行下一个计算（DrRacket的交互窗口就是这么做的）。这三者其实都只是恒等函数的变体。所以，我们定义以下函数辅助测试：

```
(define (run c) (c identity))
```

例如，

```
(test (run (cps 3))                         3)
(test (run (cps ((lam ()    5)        )))      5)
(test (run (cps ((lam (x)   (* x x)) 5)))     25)
(test (run (cps (+ 5 ((lam (x) (* x x)) 5)))) 30)
```

也可以测试之前的Web程序：

```
(run (cps (display (+ (read-number "First")
                      (read-number "Second")))))
```

为了避免你迷失在众多代码之中，我强调一下这里的重点：我们恢复了代码的结构。换种说法，即借由恰当的嵌套表达式以及帮助将其翻译以使其可以和底层API协作的代码的编译器（本例中即CPS转换程序），我们得以使用直述的风格（**direct style**）编写程序。这正是优秀的编程语言所应做的！

14.2.2 例子的转化

让我们来看看上面的例子是怎么转换的。你可以手工操作，也可以采取简单的办法，用DrRacket的Macro Stepper（宏步进器）完成。【注释】放入run函数传入的恒等函数，我们得到：

```
(lambda (k)
  ((lambda (k)
     ((lambda (k)
        ((lambda (k)
           (k "First")) (lambda (pv)
                          (read-number/suspend pv k))))
      (lambda (lv)
        ((lambda (k)
           ((lambda (k)
              (k "Second")) (lambda (pv)
                              (read-number/suspend pv k))))
         (lambda (rv)
           (k (+ lv rv)))))))
   (lambda (ov)
     (k (display ov)))))
```

> 这里，为了获取的Macro Stepper的全部功能，请使用`#lang racket`语言。

什么！这和我们手写的版本完全不同！

实际上，这个程序中充满了所谓的管理性lambda（administrative lambda），由我们所用的CPS算法引入。【注释】请不用担心！如果我们逐一调用这些lambda，完成替代，那么——

思考题

> 完成此步。

——这个程序会简化为

```
(read-number/suspend "First"
                     (lambda (lv)
                       (read-number/suspend "Second"
                                            (lambda (rv)
                                              (identity
                                               (display (+ lv rv)))))))
```

这正是我们想要的。

> 设计更好的CPS算法，消除不必要的管理性lambda，是个研究前沿问题。

14.2.3 在核心中实现

在研究了通过去语法糖实现CPS之后，我们应该问问，是否可将其以放在核心中。

回想一下，我们说过CPS适用于任何程序。有一个我们特别感兴趣的程序：解释器。显然，我们可以将CPS转换应用于其上，从而获得事实上的continuation。

首先，这里使用函数来表示闭包较为方便（译注，12.1节）。我们让解释器读入多读入一个参数，该参数读入值（需要传给continuation的那些值）并最终返回它们：

```
<cps-interp> ::=  ;cps解释器

    (define (interp/k [expr : ExprC] [env : Env] [k : (Value -> Value)]) : Value
      <cps-interp-body>)  ;cps解释器主体
```

对于简单的情况，我们不直接返回值，而是将其传递给continuation参数即可：

```
<cps-interp-body> ::=

    (type-case ExprC expr
      [numC (n) (k (numV n))]
      [idC (n) (k (lookup n env))]
      <cps-interp-plusC-case>
      <cps-interp-appC-case>
      <cps-interp-lamC-case>)
```

（请注意，multC的处理完全类似于plusC。）

还是从简单的情况开始，plusC。第一步我们解释左子表达式。该计算的continuation进行右子表达式的解释。这个计算的continuation对结果求和。求和的结果怎么处理？在interp中，它被返回，返回到那个调用解释plusC的计算。请记住，现在我们不再返回值；反之，我们将其传给continuation：

```
<cps-interp-plusC-case> ::=

    [plusC (l r) (interp/k l env
                       (lambda (lv)
                         (interp/k r env
                                   (lambda (rv)
                                     (k (num+ lv rv))))))]
```

习题

> 实现multC。

还剩下两种相互关联的情况，它们相对更难些。

对于函数调用，还是需要解释两个子表达式，然后将结果的闭包应用于参数。不过，我们已经说好了，每个调用都需要带上continuation参数。因此，必须更新一下值的定义：

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value (Value -> Value) -> Value))])
```

接下来必须决定传给它啥continuation。对于函数调用，就是传入解释器的continuation：

```
<cps-interp-appC-case> ::=

    [appC (f a) (interp/k f env
                      (lambda (fv)
                        (interp/k a env
                                  (lambda (av)
                                    ((closV-f fv) av k)))))]
```

最后处理lamC的情况。和以前一样，我们必须使用lambda创建closV。不过，这个函数需要两个参数：实际的参数和调用的continuation。关键的问题是，后者该是什么？

有两个选择。k表示静态的continuation：在闭包创建位置的那个continuation。不过，我们想要的是在闭包调用之处的continuation，也就是动态的continuation。

```
<cps-interp-lamC-case> ::=

    [lamC (a b) (k (closV (lambda (arg-val dyn-k)
                        (interp/k b
                                  (extend-env (bind a arg-val)
                                              env)
                                  dyn-k)))))]
```

要测试这个修改后的解释器，我们需要用某个初始continuation调用interp/k。这个子程序表示的是无需任何其他计算。自然的选择是恒等函数：

```
(define (interp [expr : ExprC]) : Value
  (interp/k expr mt-env
            (lambda (ans)
              ans)))
```

为了强调这只是interp/k的顶层接口，interp放弃了环境参数，自动传递空环境给interp/k。如果需要特别确定没有意外地递归使用这个函数，我们可以在其最后插入一个对error的调用，以防止它返回，或者其返回值被使用。

14.3 生成器

现在许多编程语言都拥有生成器（generator）这一概念。生成器类似于函数，可以被调用。区别在于，常规函数总是从头开始执行，生成器从最后一次停止的地方恢复。当然，这意味着生成器需要“在完成之前退出”的概念。这就是所谓的**yield**（让位），即把控制权归还给调用者。

14.3.1 各种设计

生成器有许多不同的变体。可以想见，不同之处在于如何进入和退出生成器：

- 在某些语言中，生成器是一种对象，需要和其他对象一样实例化，恢复其执行是通过调用方法（例如Python中的next）。在其他语言中，生成器则类似于函数，而且重入是通过像函数一样调用。【注释】

- 在某些语言中，让位操作——例如Python的yield——只能在生成器的语法主体中使用。在其他语言中，例如Racket，yield是在生成器主体中被绑定的、可调用的值，正由于它是值，它可以被抽象的传递、存储于数据结构中，等等。

> 在有些语言中，除了普通的函数，其他值也可以用做调用，所有这些值被统称为可调用值（applicable）。

Python的设计代表了一种极端，生成器是任何包含关键字**yield**的函数。此外，Python的yield不能作为参数传递给另一个函数，由该函数代理来执行让位。

还有个关于命名的小问题。在许多支持生成器的语言中，让位指令就是字面上的yield：要么是关键字（如Python），要么是绑定为可调用值的标识符（如在Racket中）。还有种可能，生成器的用户必须在生成器表达式中指明让位指令的名字。【注释】也就是说，生成器是这样的

```
(generator (yield) (from)
           (rec (f (lam (n)
                     (seq
                       (yield n)
                       (f (+ n 1)))))
             (f from)))
```

但是等价的写法

```
(generator (y) (from)
           (rec (f (lam (n)
                     (seq
                       (y n)
                       (f (+ n 1)))))
             (f from)))
```

如果这个让位指令实际上是值，那么用户也可以这样抽象地使用：

```
(generator (y) (from)
           (rec (f (lam (n)
                     (seq
                       ((yield-helper y) n)
                       (f (+ n 1)))))
             (f from)))
```

其中yield-helper会去调用让位指令。

实际上还有两个设计上的决定：

1. yield是声明还是表达式？在许多语言中，它是表达式，这意味着它有值：在恢复生成器时提供的值。这使得生成器更加灵活，因为生成器的使用者可以使用参数来改变生成器的行为，而不是被迫使用状态来传达所需的改变。
2. 生成器执行结束时会发生什么？在很多语言中，生成器会产生异常来表示完成。

> 奇怪的是，Python在对象中期望用户来确定self或this的名称，但是它没有为yield提供相同的灵活性，因为这是唯一确定哪些函数是生成器的方式！

14.3.2 实现生成器

要实现生成器，有效的方式是使用我们的CPS宏语言。先来确定这个设计决定的意义。我们用调用来表示生成器：即，要获得来自生成器的下一个值，是通过将其应用于任何必要的参数来完成的。类似的，让位指令也是可调用的值，并且还是表达式。虽然我们已经研究过宏如何自动捕获名称（译注：13.5节），但是简单起见我们还是明确给出让位指令的名称好了。最后，当生成器执行完成时，我们会报错。

生成器如何工作？ 要yield，生成器必须

- 记住它现在执行到哪里，
- 知道应该返回到调用者的哪里。

而当生成器被调用时，它应该

- 记住它的调用者执行到哪里，
- 知道它应该返回到其主体内的哪里。

请注意调用与让位之间的对偶。

你可能猜到了，这些“哪里”就是continuation。

我们来逐步实现生成器，这相当于添加一条cps宏的规则。先写下模式的头部：

```
<cps-macro-generator-case> ::=  ;CPS宏，生成器子句

    [(_ (generator (yield) (v) b))
     (and (identifier? #'v) (identifier? #'yield))
     <generator-body>]  ;生成器主体
```

主体第一部分很简单：CPS中的所有代码都需要先读入continuation，而且由于生成器是值，所以这个值要被传给continuation：

```
<generator-body> ::=  ;生成器主体

    #'(lambda (k)
        (k <generator-value>))  ;生成器的值
```

下一步要处理生成器的核心了。

回忆一下，生成器是可调用的值。这就是说，它可以被放在函数调用的位置，因此它必须具有与函数相同的“接口”：函数有两个参数，第一个是值，第二个是调用位置的continuation。这个子程序应该做什么？我们刚刚描述过这个。首先，生成器必须记住它的调用者正在执行的地方，这正是调用位置的continuation；“记住”这里最简单的意思是“必须保存在状态中”。然后，生成器应该返回到它之前所在的地方，即它自己的continuation，这个显然必须被保存过。因此，这里可调用值的核心是：

```
<generator-core> ::=  ;生成器的核心

    (lambda (v dyn-k)
      (begin
        (set! where-to-go dyn-k)
        (resumer v)))
```

这里，where-to-go记录了调用者的continuation，让位时恢复；resumer是生成器的本地continuation。让我们考虑一下它们的初始值是什么：

- where-to-go没有初始值（因为生成器尚未被调用），所以如果它被调用，需要抛出错误。幸运的是，这个错误永远不会发生，因为第一次进入生成器时会对where-to-go赋值，所以这个错误只是防范实现中出现bug。
- 最初，生成器的其余部分是整个生成器，所以resumer应该被绑定到b（的CPS）。它的continuation是什么？是整个生成器的continuation，即当生成器结束时该做啥。我们已经

讨论过，这里也应该给出错误（区别是，在这种情况下错误确实会发生，如果生成器被要求产生比它配备的更多的值）。

还需要绑定yield。正如我们已经指出的，它对称于生成器的恢复：将本地continuation保存在resumer中，然后通过调用where-to-go返回。

把这些片段放到一起，我们得到：

```
<generator-value> ::=  ;生成器的值

    (let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
      (letrec([resumer (lambda (v)
                         ((cps b) (lambda (k)
                                    (error 'generator "fell through"))))]
               [yield (lambda (v gen-k)
                        (begin
                          (set! resumer gen-k)
                          (where-to-go v)))])
        <generator-core>))
```

思考题

> 为什么这里使用let和letrec，而不只用let？

请注意这些代码片段之间的依赖关系。where-to-go不依赖于resumer或yield。yield显然依赖于where-to-go和resumer。但是，为什么resumer和yield相互引用呢？

思考题

> 试试不这么做。

你可能会遗漏的巧妙依赖是，resumer中包含b，生成器的主体，它可能包含对yield的引用。因此，它需要包含退位指令的绑定。

练习

> 生成器与协程（coroutine）和线程（thread）有什么不同？使用类似的策略来实现协程和线程。

14.4 Continuation和堆栈

虽然看上去不明显，但是CPS转换实际上对程序执行的栈（译注，调用栈）本质提供了深入的了解。首先要理解的是，continuation实际上就是栈本身。这可能看起来很奇怪，因为堆栈是底层的机器实现，而continuation看似复杂。那么栈到底是什么呢？

- 栈是还有待完成的计算的记录。continuation也是。
- 栈传统上被认为是栈帧（stack frame）的链表。也就是说，每个帧都引用该帧完成后剩余的帧。类似地，每个continuation都是个小程序，其中引用——因此包含——自己的continuation。如果为程序指令选择不同的表示形式，将其与闭包的数据结构表示相结

合，我们将得到一种与计算机堆栈基本相同的continuation表示法。

- 每个栈帧中还存储了函数的参数。continuation的子程序表示法隐式地管理了此项信息，明确地由数据结构（绑定）表示。
- 栈帧中还有“局部变量”的空间。continuation原则上也是如此，尽管我们使用宏实现本地绑定，因此相当于将一切都还原成函数参数。然而从概念上讲，其中一些是“真实的”函数参数，而另一些是通过宏变成函数参数的局部绑定。
- 栈引用了堆，但没有内含堆。因此，堆中的变化在不同的栈帧都是可见的。同样地，闭包中引用了贮存，但不内含贮存，所以对贮存的修改在不同闭包中都是可见的。

因此，传统上，栈负责维护词法范围，而我们使用（静态范围的语言中的）闭包自动获得此功能。

现在我们可以研究各种子项的转换，从而解到堆栈的映射。例如，考虑函数应用的转换：

```
[(_ (f a))
 #'(lambda (k)
     ((cps f) (lambda (fv)
                ((cps a) (lambda (av)
                           (fv av k))))))]
```

该怎么“读”呢？这样：

- 我们用k表示函数调用之前的栈。
- 在对函数位置（`f`）求值时，创建新的栈帧（`(lambda (fv) ...)`）。该帧包含一个自由标识符：`k`。因此，它的闭包需要记录环境中的这个元素，即栈的其余部分。
- 栈帧的代码部分表示一旦我们获得了函数的值，剩下的工作：计算参数，执行调用，将结果返回给等待调用结果的栈：k。
- 对f的求值完成后，对a求值，这也需要创建栈帧：`(lambda (av) ...)`。该帧有两个自由标识符：k和fv。这说明：
  - 我们不再需要对函数位置求值的栈帧了，但是
  - 我们需要用临时变量记录函数位置求值的结果，它最好是函数值。
- 这第二个帧的代码部分代表也是剩下要做的事情：对参数调用函数，在等待调用结果的栈中进行。

条件指令也是同样的推理：

```
[(_ (cnd tst thn els))
 #'(lambda (k)
     ((cps tst) (lambda (tstv)
                  (if tstv
                      ((cps thn) k)
                      ((cps els) k)))))]
```

它说的是，要对条件表达式求值，我们先要创建新的栈帧。该帧中包含等待整个条件表达式值的栈。该帧根据条件表达式的值来决定，调用其子表达式之一。在判断了条件的值之后，为了求它的值而创建的帧就不再需要了，因此求值可以在k中继续。

从这个角度出发，我们可以更好的解释生成器的操作。每个生成器都有自己的私有栈，当执行超越其栈底时，我们的实现会报错。被调用时，生成器将表示“剩余程序”的栈的引用存储在where-to-go中，然后恢复自己的栈。在让位时，系统交换堆栈的引用。协程，线程和生成器在概念上都是相似的：它们都是创建“许多小堆栈”的机制，而不仅仅只是单个的全局堆栈。

14.5 尾调用

观察上面的栈模式，为当前栈添加帧，执行一些计算，最终总是返回到当前栈。特别要注意的是，在函数调用中，我们需要栈的空间来对函数求值，然后是对参数求值，但是一旦所有这些求值完成，我们就使用函数调用开始之前的栈来恢复计算。换一种说法，函数调用本身不需要消耗栈空间：我们只需要空间来计算参数。

但是，并非所有的语言都遵守或尊重这一属性。在这样做的语言中，程序员可以使用递归来获得迭代行为：即，一系列函数调用不会比没有函数调用的情况下消耗更多空间。这消除了创建特殊循环结构的需要；实际上，循环可以简单地表示为语法糖。

当然，这个属性不适用于一般情况。如果调用f来计算调用g所需的参数，那么对f的调用相对于围绕g的上下文仍然会占用空间。因此，我们需要说明表达式之间的关系：一个表达式的处于另一表达式的尾位置，如果对它的求值不需要另一表达式（求值）之外的额外空间。在我们的CPS宏中，所有使用k作为其continuation的表达式——例如，在所有子表达式求值完成之后的函数调用，或者条件表达式的then和else分支——都在其外层表达式的尾位置（也许递归地还在其外层的尾位置）。反之，所有必须创建新栈帧的表达式都不在尾位置。

有些语言对尾递归——某个函数在其函数体的尾位置调用自己——有特殊的支持。这显然是有用的，因为它使得递归得以有效地实现循环。然而，它破坏了不能被挤入单个递归函数的“循环”。例如，当实现状态机时，最方便的方法是用一组函数，每个函数代表一个状态，然后通过（尾）调用表示状态转换。把它们变成单一的递归函数会非常繁琐（并且失去了意义）。但是，如果一种语言能够识别尾调用，它就可以（和函数内调用自己一样）优化这些跨函数的调用。

Racket的实现保证尾调用不会分配额外的栈空间。有人把这称为“尾调用的优化”，但这个术语是误导性的：优化是可选性的，而某种语言是否承诺正确实现尾调用是种语义特性。程序员需要了解语言的行为方式，因为这会影响他们的编程方式。

由于这个特性，观察CPS转换之后的程序的有趣之处：其中所有的函数调用本身都是尾调用的！从本章开头的read-number/suspend例子开始，你就可以看到这点：所有待处理的计算都被放入了continuation参数。假设程序可能在任何调用中终止，等同于根本不使用任何栈空间（因为栈将会被清除）。

练习

> 程序如何在没有栈的情况下运行？

14.6 语言特性中支持continuation

了解connection和栈之间的这种关联之后，现在可以回过头讨论函数的处理：我们忽略了在创建闭包时的continuation，而只使用了在闭包调用时的continuation。当然，这对应于普通的函数行为。但现在我们可以问，如果我们用创建时的connection呢？这等同于，在“程序”创建时保存对栈的（副本）的引用，然后在调用函数时忽略动态的求值，返回到函数创建点。

实际上，我想说的是，让lambda保持不变，而给我们的语言提供新的、对于与这种行为的指令：

```
<cps-macro-let/cc-case> ::=  ;cps宏

    [(_ (let/cc kont b))
     (identifier? #'kont)
     #'(lambda (k)
         (let ([kont (lambda (v dyn-k)
                       (k v))])
           ((cps b) k)))]
```

这说的是，两种情况下，控制都将返回到直接包含let/cc的表达式：要么通过正常返回（因为主体b的continuation是k），要么通过更有意思的方式，调用continuation，这会丢弃动态的continuation dyn/k，简单地忽略它直接返回到k。

最简单的测试是：

```
(test (run (cps (let/cc esc 3)))
      3)
```

这证实了，如果我们从不使用continuation，那么对主体的求值就好像let/cc根本不存在一样（因为`((cps b) k)`）。如果我们使用它，传给continuation的值返回到创建点：

```
(test (run (cps (let/cc esc (esc 3))))
      3)
```

当然，这个例子揭露的还不够，不过考虑这个：

```
(test (run (cps (+ 1 (let/cc esc (esc 3)))))
      4)
```

这证实了加法会实际执行。那么动态的continuation呢？

```
(test (run (cps (let/cc esc (+ 2 (esc 3)))))
      3)
```

这表明加2不会发生，即动态continuation确实被忽略了。为了确保创建位置的continuation被保留，请观察：

```
(test (run (cps (+ 1 (let/cc esc (+ 2 (esc 3))))))
      4)
```

从这些例子中，你可能已经注意到熟悉的模式：esc在这里的表现类似于异常。也就是说，如果你不抛出异常（在这里，调用continuation）它就好像不在那里，但是如果你抛出异常，所有未完成的中间计算都将被忽略，计算返回到异常创建点。

练习

> 使用let/cc和宏实现异常的抛出和捕获机制。

然而，这些例子只用到了最浅层的（let/cc的）能力，因为这里调用点处的continuation总是创建点处的continuation的扩展：即后者在栈中比前者更早。然而，没有任何东西要求k和dyn-k之间存在相关。它们实际上可以是无关的，这意味着它们可以是两个独立的栈，所以我们可以用它轻松地实现栈切换功能。

练习

> 为了真正与lambda类似，我们应该引入如下展开的构造，称其为cont-lambda好了：
>
> ```
> [(_ (cont-lambda (a) b))
>  (identifier? #'a)
>  #'(lambda (k)
>      (k (lambda (a dyn-k)
>           ((cps b) k))))]
> ```
>
> 为什么我们没有这么做呢？从两方面考虑，静态类型的角度，还有，我们如何使用这个构造来构建上述类似于异常的行为。

14.6.1 用语言表达

用我们的小玩具语言编写程序很快会变得令人沮丧。幸运的是，Racket已经提供了叫做call/cc的构造，用来操作continuation。call/cc是单参数的函数，其参数本身又是单参数的函数，Racket会将当前continuation传给它进行调用，而当前continuation也是单参数的子程序。能理解吗？

幸运的是，我们可以用call/cc轻松地将let/cc实现为宏，然后用它来编写程序。这样：

```
(define-syntax let/cc
  (syntax-rules ()
    [(let/cc k b)
     (call/cc (lambda (k) b))]))
```

之前的所有测试仍然通过：

```
(test (let/cc esc 3) 3)
(test (let/cc esc (esc 3)) 3)
(test (+ 1 (let/cc esc (esc 3))) 4)
(test (let/cc esc (+ 2 (esc 3))) 3)
(test (+ 1 (let/cc esc (+ 2 (esc 3)))) 4)
```

14.6.2 定义生成器

现在我们可以创建有趣的抽象了。比如，让我们来编写生成器。之前我们需要将表达式CPS转化，并传递continuation，现在都可以通过call/cc自动完成。因此，当需要目前的continuation时，我们都可以简单地召唤它而无需改变程序。所以，额外的 `...-k` 参数都会消失，在同一个地方可以用let/cc捕获相同的continuation：

```
(define-syntax (generator e)
  (syntax-case e ()
    [(generator (yield) (v) b)
     #'(let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
         (letrec ([resumer (lambda (v)
                             (begin b
                                    (error 'generator "fell through")))]
                  [yield (lambda (v)
                           (let/cc gen-k
                             (begin
                               (set! resumer gen-k)
                               (where-to-go v))))])
           (lambda (v)
             (let/cc dyn-k
               (begin
                 (set! where-to-go dyn-k)
                 (resumer v))))))]))
```

请观察这段代码和去语法糖到CPS代码实现的生成器之间的密切相似性。具体而言，我们去掉了额外的continuation参数，用let/cc调用替换它们，这些调用能捕获完全相同的continuation。其余的代码基本不变。

练习

> 如果我们将（两处）let/cc和赋值移到begin内的第一个语句，会发生什么呢？

例如，我们可以编写从初始值向上迭代的生成器：

```
(define g1 (generator (yield) (v)
                      (letrec ([loop (lambda (n)
                                       (begin
                                         (yield n)
                                         (loop (+ n 1))))])
                        (loop v))))
```

其行为是：

```
> (g1 10)
10
> (g1 10)
11
> (g1 0)
12
>
```

因为（生成器）主体只引用了初始值，调用yield所返回的值被忽略，所以在后续调用传入的值不起作用。相反，考虑这个生成器：

```
(define g2 (generator (yield) (v)
                      (letrec ([loop (lambda (n)
                                       (loop (+ (yield n) n)))])
                        (loop v))))
```

在第一次调用时，它返回输入的值。在此后的调用中，该值被加到后续调用生成器所提供的值上。换一种说法，该发生器累加它的所有输入值：

```
> (g2 10)
10
> (g2 15)
25
> (g2 5)
30
```

练习

> 现在我们已经使用call/cc和let/cc实现了生成器，请用它们实现协程和线程。

14.6.3 定义线程

完成生成器之后，我们再做个类似的功能：线程。具体来说，我们希望能够编写如下的程序：

```
(define d display) ;;有用的简写

(scheduler-loop-0
 (list
  (thread-0 (y) (d "t1-1  ") (y) (d "t1-2  ") (y) (d "t1-3 "))
  (thread-0 (y) (d "t2-1  ") (y) (d "t2-2  ") (y) (d "t2-3 "))
  (thread-0 (y) (d "t3-1  ") (y) (d "t3-2  ") (y) (d "t3-3 "))))
```

输出应该是：

```
t1-1  t2-1  t3-1  t1-2  t2-2  t3-2  t1-3 t2-3 t3-3
```

我们来创建必要的组件实现此功能。

我们先来定义线程调度器。它读入“线程”的链表，我们假设线程的接口读入continuation，并最终将控制返回给此continuation。每当调度器重新激活某个线程时，都会向其提供continuation。调度器可以用简单的循环（round-robin）方式选择线程，也可以使用更复杂的算法；这里我们不关心如何选择的细节。

类似于生成器，我们假定让位由调用用户命名的子程序完成，例如这里的y。我们也可以使用名称捕获（译注，13.5节）自动绑定其名称，比如yield。

这里的要点的是，请注意让位由线程系统的用户手动控制。这就是所谓的协作式多任务处理（cooperative multitasking）。相反，我们可以选择通过生成定时器或其他内在机制自动触发让位，而无需用户许可。这被称为抢占式多任务处理（preemptive multitasking）（因为系统从线程中“抢占”——也就是夺取了——控制权）。虽然这种区别对于构建系统来说是非常重要的，但从设置continuation的角度来看，这并不重要。

练习

> 在完成协作式多任务之后，实现抢占式多任务。哪里需要修改？

陈述了这些限制，我们可以着手编写调度器了。它读入线程的链表，只要还有剩下的线程就继续执行。每次，它将线程应用于continuation，这个continuation表示返回到调度器并继续下一个线程：

```
(define (scheduler-loop-0 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
     (begin
       (let/cc after-thread ((first threads) after-thread))
       (scheduler-loop-0 (append (rest threads)
                                 (list (first threads)))))]))
```

当接收线程调用绑定到after-thread的continuation时，控制返回到begin序列中第一个语句的结尾。因此，提供给continuation的值会被忽略（所以可以用任何值；我们选择用 `'dummy` ，以便其莫名出现时方便地发现问题）。将最近调用的线程附加到线程表的末尾（即，将该链表视为循环队列）之后，控制将继续调度器循环的其余部分。

接下来我们定义线程。我们说过，它是单参数的函数，参数就是调度器的continuation。由于线程需要能恢复，也就是从停止的地方继续，所以它必须存储上次执行到的位置：我们将其称为thread-resumer。起初thread-resumer是整个线程体，但在后续的实例中，它将是continuation：调用yield的continuation。于是，我们得到如下的框架：

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))])
       (lambda (sched-k)
         (thread-resumer 'dummy)))]))
```

还剩下yielder没实现。它是无参数的函数，将线程的continuation存入thread-resumer，然后用 `'dummy` 调用调度器的continuation。不过，调用哪个调度器的continuation呢？不是线程初始化时传入的那个，而是最新的那个。因此，我们必须以某种方式将sched-k中的值“thread”（译注，传递）给yielder。有很多种方式可以实现，但最简单的，也许是最暴力的方式是，简单地为每个线程恢复重建yielder，总是包含sched-k的最新值：

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))]
              [yielder (lambda () (error 'yielder "nothing here"))])
       (lambda (sched-k)
         (begin
           (set! yielder
                 (lambda ()
                   (let/cc thread-k
                     (begin
                       (set! thread-resumer thread-k)
                       (sched-k 'dummy)))))
           (thread-resumer 'tres))))]))
```

将这些放到一起运行，我们得到：

```
t1-1  t2-1  t3-1  t1-2  t2-2  t3-2  t1-3 t2-3 t3-3
```

嘿，这就是我们想要的！但是运行继续：

```
t1-3 t2-3 t3-3 t1-3 t2-3 t3-3 t1-3 t2-3 t3-3
```

嗯。

怎么回事？恩，我们并没有说明当线程运行结束时需要怎么处理。实际上，控制只是返回到线程调度器，调度器将线程追加到队列的末尾，然后，当线程再次到达队列的头部时，控制从之前存储的那个continuation中恢复：对应于打印第三个值。打印，控制返回，线程被追加到队尾……无限循环。

显然，在线程终止时，我们需要通知线程调度器，这样调度器可以将其从线程队列中移除。我们创建简单的数据类型来表示该信号：

```
(define-type ThreadStatus
  [Tsuspended]
  [Tdone])
```

（当然，在真实的系统中，这些状态消息也可以带上和计算相关的值。）那么我们必须修改调度器，实际检查和使用这些值：

```
(define (scheduler-loop-1 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
     (type-case ThreadStatus (let/cc after-thread ((first threads) after-thread))
       [Tsuspended () (scheduler-loop-1 (append (rest threads)
                                                (list (first threads))))]
       [Tdone () (scheduler-loop-1 (rest threads))])]))
```

线程的表示中有两个地方需要修改：中间返回的时候它必须传Tsuspended给调度器的continuation，终止时传Tdone。哪里是终止呢？在执行完线程体代码 `b ...` 之后。最后，请注意和退位一样，终止程序必须也使用最新的调度器continuation。因而：

```
(define-syntax thread-1
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...
                                       (finisher)))]
              [finisher (lambda () (error 'finisher "nothing here"))]
              [yielder (lambda () (error 'yielder "nothing here"))])
       (lambda (sched-k)
         (begin
           (set! finisher
                 (lambda ()
                   (let/cc thread-k
                     (sched-k (Tdone)))))
           (set! yielder
                 (lambda ()
                         (let/cc thread-k
                           (begin
                             (set! thread-resumer thread-k)
                             (sched-k (Tsuspended))))))
           (thread-resumer 'tres))))]))
```

用scheduler-loop-1和thread-1替换scheduler-loop-0和thread-0，重新运行前面的示例程序，就能得到正确的输出。

14.6.4 更好的Web编程指令

最后，我们回过头看看read-number：请注意，如果运行服务器程序的语言有call/cc，我们就不必CPS整个程序，而是可以简单地捕获当前continuation，将其保存在哈希表中，从而使程序结构保持不变。

15 静态地检查程序中的不变量：类型

当程序变得更大或者更为复杂时，程序员希望能有工具帮助他们描述、验证程序中的不变量。顾名思义，不变量指的就是关于程序组成元素的那些不会发生改变的陈述。例如，当我们在静态类型语言中写下 `x : number` 时，表示 x 中存放的总是数，程序中依赖 x 的部分都可以认定它是数的这个事实不会改变。我们将会看到，类型只是我们想要陈述的各类不变量中的一种，静态类型检测——一个分支众多的技术家族——也只是用于控制不变量的众多方法中的一个。

15.1 静态类型规则

本章我们将专注于静态类型检查：即在程序执行前检查（声明的）类型。之前使用的静态类型语言已经让我们积攒了一些这种形式程序的经验。我们将探索类型的设计空间及这些设计中的权衡取舍。尽管类型是控制不变量的一种非常强大且有效的方法，最后我们还是会考察一些其它可用的技术。

考虑下面这段静态语言写就的程序：

```
(define (f [n : number]) : number
  (+ n 3))

(f "x")
```

程序开始执行前我们就会得到一个静态类型错误。使用普通 Racket 写就的同样的程序（去除类型注解）只会在运行时出错：

```
(define (f n)
  (+ n 3))

(f "x")
```

练习题

> 如何判断错误是在程序执行前还是运行时抛出的？

考虑下面这段 Racket 程序：

```
(define f n
  (+ n 3))
```

它也是在程序执行前就遇到错误——语法解析错误——终止。尽管我们认为语法解析和类型检查有所不同——通常是因为类型检测是针对已经被解析好的程序做的——但是将语法解析看作一种最简单形式的类型检查也很有用：它（静态地）判定程序是否遵守某个上下文无关

语法。随后，类型检查判定它是否遵守某个上下文相关（或者一个更丰富的）语法。简而言之，类型检查从某种程度上看是语法解析的泛化，它们都是通过语法控制程序遵循指定的规则。

15.2 关于类型的经典看法

我们先介绍传统的包含类型的核心语言；然后我们将探索其扩展和变种。

15.2.1 简单的类型检查器

要定义类型检查器，我们先需要就两件事达成一致：我们静态类型核心语言的语法，对应的类型的语法。

先回到我们之前实现过的函数作为值的那一版语言，其中并不包含赋值等其它稍复杂的东西（后面将讲到添加其中的一些）。我们需要为该语言添加类型注解。按惯例，我们不对常量或基本操作（如加法）强加类型注释；相反，我们把类型注释加在函数或方法的边界上。在本章讨论的过程中，我们将探讨为什么这么做。

鉴于此决定，我们静态类型的核心语言变成了：

```
(define-type TyExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [appC (fun : TyExprC) (arg : TyExprC)]
  [plusC (l : TyExprC) (r : TyExprC)]
  [multC (l : TyExprC) (r : TyExprC)]
  [lamC (arg : symbol) (argT : Type) (retT : Type) (body : TyExprC)])
```

每个函数都添加了其参数及返回值类型的注解。

现在我们需要对类型语言作出选择。我们遵从传统定义，即类型是一组值的集合的抽象。我们的语言中有两类值：

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : TyExprC) (env : Env)])
```

因此我们有两种类型：数和函数。

即使数类型也并不那么简单直接：数类型应该记录何种信息？大部分语言中，实际上有很多数类型，甚至没有哪个类型表示“数”。然而，我们忽略了数的层级结构（译注，第三章），对于我们来说有一种数的类型足矣。这样决定之后，我们是否需要记录哪种数的信息？ 原则上可以，但这样我们很快就会遇到可判定性问题。

至于函数，我们有更多信息：参数的类型，返回值的类型。我们不妨记录下这些信息，除非事后证实这些信息没有用处。结合这些，我们得出这样的类型的抽象语言：

```
(define-type Type
  [numT]
  [funT (arg : Type) (ret : Type)])
```

既然已经确定了语言中项和类型的结构，接下来我们来确定语言中哪些算是类型错误（并且，如果程序中不包含这里列出的类型错误，它就会通过类型检查）。显然有三种形式的类型错误：

- `+` 的参数不是数，即不是 `numT` 。
- `*` 的参数不是数。
- 函数调用时函数位置的表达式不是函数，即不是 `funT` 。

思考题

> 还有其它形式的类型错误吗？

事实上我们遗漏了一个：

- 函数调用时实参的类型和函数形参的类型不一致。

我们的语言中的所有其他程序似乎都应该通过类型检查。

关于类型检查器的签名，初步设想，它可以接受表达式作为参数，返回布尔值指明该表达式是否通过检查。由于我们知道表达式中包含标识符，所以很显然我们还需要一个类型环境，它将名字映射到类型，类似于我们之前用到的值环境。

练习题

> 定义与类型环境相关的数据类型以及函数。

于是，我们开始写下的程序结构大致是这样：

```
<tc-take-1> ::=  ;;类型检查，第一次尝试

    (define (tc [expr : TyExprC] [tenv : TyEnv]) : boolean
      (type-case TyExprC expr
        <tc-take-1-numC-case>
        <tc-take-1-idC-case>
        <tc-take-1-appC-case>))
```

正如上面程序中列出的要处理几种情形所表明的，这种方法行不通。我们很快将知道这是为什么。

首先处理简单的情形：数。单独的一个数能通过类型检查吗？显然可以；它所处的上下文可能想要的不是数类型，但是这种错误应该在其它地方被检查出。因此：

```
<tc-take-1-numC-case> ::=

    [numC (n) true]
```

下面处理标识符。如何判断标识符是否通过类型检查呢？同样，就其自身来说，如果是绑定标识符，总是通过检查的；它可能不是上下文要求的那种类型，但是这种错误应该在其它地方检查。因此，我们得出：

```
<tc-take-1-idC-case> ::=

    [idC (n) (if (lookup n tenv)
                 true
                 (error 'tc "not a bound identifier"))]  ;不是绑定标识符
```

上面的代码你可能感觉不太对：如果标识符未绑定的话，`lookup`会抛出异常，因此没必要再去重复处理该情况（事实上，代码永远不会执行到`error`调用那个分支）。但是让我们先继续。

下面来处理函数调用。我们应该首先检查函数位置，确定它是个函数，然后确保实际参数的类型和该函数定义时声明的形式参数类型相同。例如，函数可能需要参数是数，但调用给的是个函数，或者反之，在这两种情况下，我们都需要防止错误的函数调用。

代码该怎么写？

```
<tc-take-1-appC-case> ::=

    [appC (f a) (let ([ft (tc f tenv)])
                  ...)]
```

对于`tc`的递归调用只能让我们知道函数位置是否通过类型检查。如果它通过了，怎么知道它具体是什么类型的呢？如果是个简单的函数定义的话，我们可以直接从语法上取得其参数和返回值的类型。但是如果是个复杂的表达式，我们就需要一个函数能计算出表达式类型。当然，只有这个表达式是个类型正确的表达式时，该函数才能返回类型结果；否则的话它将不能得出正确的结果。换句话说，“类型检查”是“类型计算”的一种特殊情形！因此，我们应该增强`tc`的归纳不变量：即，不仅仅返回表达式是否能通过类型检查，而是返回表达式的类型。事实上，只要有返回值，就说明该表达式通过了类型检查；否则它会抛出错误。

下面我们来定义这个更完善的类型“检查器”。

```
<tc> ::=

    (define (tc [expr : TyExprC] [tenv : TyEnv]) : Type
      (type-case TyExprC expr
        <tc-numC-case>
        <tc-idC-case>
        <tc-plusC-case>
        <tc-multC-case>
        <tc-appC-case>
        <tc-lamC-case>))
```

现在填充具体实现。数很简单：它的类型就是数类型。

```
<tc-numC-case> ::=

    [numC (n) (numT)]
```

与之相似，标识符的类型从环境中查询得到（如果其未被绑定则会抛出错误）。

```
<tc-idC-case> ::=

    [idC (n) (lookup n tenv)]
```

到此，我们可以观察到该类型检查器与解释器之间的一些异同：对于标识符，两者做的事情其实一样（只不过这里返回的是标识符的类型而不是一个实际的值），对于数的情况，这里返回了抽象的“数”而不是具体的数。

下面考虑加法。必须确保其两个子表达式都具有数类型；如果满足该条件，则加法表达式本身返回的是数类型。

```
<tc-plusC-case> ::=

    [plusC (l r) (let ([lt (tc l tenv)]
                       [rt (tc r tenv)])
                   (if (and (equal? lt (numT))
                            (equal? rt (numT)))
                       (numT)
                       (error 'tc "+ not both numbers")))] ;+不都是数
```

通常在处理完加法的情形之后，对于乘法我们就一笔带过了，但是这里显式处理一下它还是很有教益的：

```
<tc-multC-case> ::=

    [multC (l r) (let ([lt (tc l tenv)]
                       [rt (tc r tenv)])
                   (if (and (equal? lt (numT))
                            (equal? rt (numT)))
                       (numT)
                       (error 'tc "* not both numbers")))] ;*不都是数
```

思考题

> 看出其中的区别了吗？

是的，基本上完全没区别！（仅有的区别是在 `type-case` 时使用的分别 `multC` 和 `plusC` ，以及错误提示信息稍有不同）。这是因为，从（此静态类型语言）类型检查的角度来说，加法和乘法没有区别，更甚，任意接受两个数作为参数返回一个数的函数都没有区别。

注意到代码解释和类型检查之间另一个不同点。它们的参数都得是数。解释器返回加或者乘它们得到的确切数值，但是类型检查器并不在乎具体的数值：因此该表达式的计算结果（ `(numT)` ）是个常数，两种情形返回都是该常数。

最后还剩下两个难一点的情形：函数调用和函数。我们已经讨论过怎么处理函数调用：计算函数以及参数表达式的值；确保函数表达式为函数类型；检查参数类型和函数形参类型相容。如果这些条件满足，函数调用得到的结果类型就是函数体的类型（因为运行时最终的返回值就是计算函数体得到的值）。

```
<tc-appC-case> ::=

    [appC (f a) (let ([ft (tc f tenv)]
                      [at (tc a tenv)])
                  (cond
                    [(not (funT? ft))
                     (error 'tc "not a function")] ;不是函数
                    [(not (equal? (funT-arg ft) at))
                     (error 'tc "app arg mismatch")] ;app参数不匹配
                    [else (funT-ret ft)]))]
```

最后还剩下函数定义。函数有一个形参，函数体中一般会用到；除非它被绑定到环境中，不然函数体应该不太可能通过类型检查。因此我们需要扩展类型环境，添加形参与其类型的绑定，然后在扩展后的环境中检查函数体。最终计算得到的函数体类型必须和函数定义中指定的函数返回值类型相同。如果满足了这些，该函数的类型就是指定参数类型到函数体类型的函数。

练习题

> 上面说的“不太可能通过类型检查”是什么意思？

```
<tc-lamC-case> ::=

    [lamC (a argT retT b)
          (if (equal? (tc b (extend-ty-env (bind a argT) tenv)) retT)
              (funT argT retT)
              (error 'tc "lam type mismatch"))] ;λ类型不匹配
```

注意到解释器与类型检查器另一个有趣的不同点。解释器中，函数调用负责计算参数表达式的值，扩展环境，然后对函数体求值。而这里，函数调用的情形中的确也检查了参数表达式，但是没有涉及到环境的处理，直接返回了函数体的类型，而没有遍历它。对函数体的遍历检查过程实际是在检查函数定义的过程中进行的，因此环境也是在这个地方才实际被扩展的。

15.2.2 条件语句的类型检查

考虑为上面的语言添加条件语句，即使最简单的 if 表达式都会引入若干设计抉择。这里我们先讨论其中的两个，后面会回过头讨论其中的一个。

1. 条件表达式的类型应该是什么？某些语言中它必须计算得到布尔值，这种情况下需要为我们的语言添加布尔值类型（这可能是个好主意）。其它语言中，它可以是任意值，某些值被认为是“真值”，其它的则被视为“假值”。
2. `then-` 和 `else-` 两个分支之间的关系应该是什么呢？一些语言中它们的类型必须相同，

> 因此整个 if 表达式有一个确定无歧义的类型。其它语言中，两个分支可以有不同的类型，这极大地改变了静态类型语言的设计和它的类型检查器，而且也改变了编程语言本身的性质。

练习题

> 为该静态类型语言添加布尔值。至少需要添加些啥？在典型的语言中还需要加什么？

练习题

> 为条件语句添加类型规则，其中条件表达式应该计算得到布尔值，且`then-`和`else-`分支必须有相同的类型，同时该类型也是整个条件语句的类型。

15.2.3 代码中的递归

现在我们已经得到了基本的编程语言，下面为其添加递归。之前我们实现过递归，可以很容易的通过去语法糖实现。这里的情况要更复杂一些。

15.2.3.1 递归的类型，初次尝试

首先尝试表示一个简单的递归函数。最简单的当然就是无限循环。我们可以仅使用函数实现无限循环吗？可以：

```
((lambda (x) (x x))
 (lambda (x) (x x)))
```

因为我们的语言中已经支持将函数作为值。

练习题

> 为什么这会构成无限循环？它是如何巧妙地依赖于函数调用的本质的？

现在我们的静态类型语言要求我们为所有函数添加类型注解。我们来为该函数添加类型注解。简单起见，假设从现在开始我们写的程序使用的语法是静态类型的表层语法，去语法糖将帮我们将其转换为核心语言。

首先注意到，我们有两个完全一样的表达式，它们互相调用。历史原因，整个表达式被称为Ω（希腊字母大写欧米茄），那两个一样的子表达式被称为ω（希腊字母小写欧米茄）。两个一样的表达式并非得是同种类型的，因为这还依赖于具体使用环境中对于不变量的定义。这个例子中，观察到 x 被绑定到ω，于是ω将出现在在`(x x)`式子的第一个和第二个部分。即，确定其中一个表达式的类型，另一个式子的类型也被确定。

那么我们就来尝试计算ω的类型；称该类型为γ。显然它是一个函数类型，而且是单参数的函数，所以它的类型必然是`φ -> ψ`这种形式的。该函数的参数是什么类型？就是ω的类型。也即，传入φ的值的类型就是γ。因此，ω的类型是γ，也即`φ -> ψ`，展开即`(φ -> ψ) -> ψ`，进

一步展开得 `((φ -> ψ) -> ψ) -> ψ` ，还可以继续下去。也就是说，该类型不能用有限的字符串写出来！

思考题

> 你注意到了我们刚做的的微妙但重要的跳跃吗？

15.2.3.2 程序终止

我们观察到，试图直接地计算Ω的类型，需要先计算γ的类型，这似乎导致了严重的问题。然后我们就得出结论：此类型不能用有限长度的字符串表示，但是这只是直觉的结果，并非证明。更奇怪的事实是：在我们迄今定义的类型系统中，根本无法给出**Ω**的类型！

这是一个很强的表述，但事实上我们可以给出更强的描述。我们目前所用的静态类型语言有一个属性，称为强归一化（strong normalization）：任何有类型的表达式都会在有限步骤后终止计算。换句话，这个特殊的（奇特的）无限循环程序并不是唯一不可获得类型的程序；任何无限循环（或潜在存在无限循环）程序都无法求得类型。一个简单的直觉说明可以帮助我们理解，任何类型——必须能被有限长度的字符串表示——只能包含有限个 `->` ，每次调用会去除一个 `->` ，因此我们只能进行有限次数的函数调用。

如果我们的程序只允许非转移程序（straight-line program），这点也无足为奇。但是，我们有条件语句，还有可以当做值任意传递的函数，通过这些我们可以编码得到任何我们想要的数据结构。然而我们仍能得到这个保证！这使得这个结果令人吃惊。

练习题

> 试着使用函数分别在动态类型和静态类型语言中编码实现链表。你看到了什么？这说明此类型系统对于编码产生了何种影响？

这个结果展示了某种更深层次的东西。它表明，和你可能相信的——类型系统只是用来避免一些程序 BUG 在运行时才被发现——相反，类型系统可能改变语言的语义。之前我们一两行就能写出无限循环，现在我们怎么都写不出来。这也表明，类型系统不仅可以建立关于某个特定程序的不变量，还能建立关于语言本身的不变量。如果我们非常需要确保某个程序将会终止，只要用该语言来写然后交由类型检查器检查通过即可。

一门语言，用其书写的所有程序都将终止，有什么用处？对于通用编程来说，当然没用。但是在很多特殊领域，这是非常有用的保证。例如，你要实现一个复杂的调度算法；你希望知道调度程序保证会终止，以便那些被调度的任务被执行。还有许多其他领域，我们将从这样的保证中受益：路由器中的数据包过滤器；实时事件处理器；设备初始化程序；配置文件；单线程 JavaScript 中的回调；甚至编译器或链接器。每种情况下，我们都有一个不成文的期望，即这些程序最终会终止。而现在我们有一个语言能保证这点——且这点是不可测试的。

> 这不是假想的例子。在Standard ML语言中，链接模块基本上就是使用这种静态类型语言来编写模块链接规范。这意味着开发人员可以编写相当复杂的抽象概念——毕竟可以将函数作为值使用——且同时链接过程被保证会终止，产生最终的程序。

15.2.3.3 静态类型的递归

这就意味着，之前我们可以只通过去语法糖来实现`rec`，现在则必须在我们的静态类型语言中显式的实现。简单起见，我们仅考虑`rec`的一种特殊形式——它涵盖了常见用法，即递归标识符被绑定到函数。因此，表层语法中，我们可能写出如下的累加函数：

```
(rec (Σ num (n num)
        (if0 n
             0
             (n + (Σ (n + -1))))) ;译注，原文如此，+应前置
  (Σ 10))
```

其中，`Σ`是函数名，`n`为其参数，`num`为函数参数以及返回值的类型。表达式`(Σ 10)`表示使用该函数计算从 10 累加到 0 的和。

如何计算这个表达式的类型？显然，求类型过程中，`n`在函数体中的类型需要绑定（但是在函数调用处就不需要了）；这一点计算函数类型的时候我们就知道了。那么`Σ`呢？显然，在检查`(Σ 10)`的类型时，它应该在类型环境中被绑定，类型必须为`num -> num`。不过，在检查函数体时，它同样需要被绑定到此类型。（还要注意，函数体返回值的类型需要和事先声明的返回类型相同。）

现在我们可以看到如何打破类型有限性的束缚。程序代码中，我们只能编写包含有限数量`->`的类型。但是，这种递归类型的规则在函数体中引用自身时复制了`->`，从而供应了无穷的函数调用。这是包含无穷箭矢的箭筒。

实现这种规则的代码如下。假设`f`被绑定到函数的名字，`aT`是函数参数的类型，`rT`为返回类型，`b`是函数体，`u`是函数的使用：

```
<tc-lamC-case> ::=

    [recC (f a aT rT b u)
          (let ([extended-env
                 (extend-ty-env (bind f (funT aT rT)) tenv)])
            (cond
              [(not (equal? rT (tc b
                                   (extend-ty-env
                                    (bind a aT)
                                    extended-env))))
               (error 'tc "body return type not correct")] ;函数体类型错误
              [else (tc u extended-env)]))]
```

15.2.4 数据中的递归

我们已经见识了静态类型的递归程序，但是它还不能使我们创建递归的数据。我们已经有一种递归数据——函数类型——但是这是内建的。现在还没看到如何创建自定义的递归数据类型。

15.2.4.1 递归数据类型定义

当我们说允许程序员创建递归数据时，我们实际在同时谈论三种东西：

- 创建新的类型
- 让新类型的实例拥有一个或多个字段
- 让这些字段中的某些指向同类型的实例

实际上，一旦我们允许了第三点，我们就必须再允许一点：

- 允许该类型中非递归的基本情况的存在

这些设计准则的组合产生了通常被称为代数数据类型（algebraic datatype）的东西，比如我们的静态语言中支持的类型。举个例子，考虑下面这个数二叉树的定义：【注释】

```
(define-type BTnum
  [BTmt]
  [BTnd (n : number) (l : BTnum) (r : BTnum)])
```

> 后面我们会讨论如何参数化类型。

请注意，如果这个新的数据类型没有名字，`BTnum`，我们将不能在`BTnd`中引用回该类型。同样地，如果只允许定义一种`BTnum`构造，那么就无法定义`BTmt`，这会导致递归无法终止。当然，最后我们需要多个字段（如`BTnd`中的一样）来构造有用、有趣的数据。换句话说，所有这三种机制被打包在一起，因为它们结合在一起才最有用。（但是，有些语言确实允许定义独立结构体。后文我们将回来讨论这个设计决策对类型系统的影响）。

我们关于递归表示的初步讨论暂告一个段落，但这里有个严重的问题。我们并没有真正解释这个新的数据类型`BTum`的来源。因为我们不得不假装它已经在我们的类型检查器中实现了。然而，为每个新的递归类型改变我们的类型检查器有点不切实际——这就好比需要为每个新出现的递归函数去修改解释器！相反，我们需要找到一种方法，使得这种定义成为静态类型语言的固有能力。后面我们会回来讨论这个问题。

这种风格的数据定义有时也被称为乘积的和，“乘”指代字段组合成不变量的方式：例如，`BTnd`的合法值是传递给`BTnd`构造器的每个字段合法值的叉乘。“和”是所有这些不变量的总数：任何给定的`BTnum`值是其中之一。（将“乘”想作“且”，“加”想作“或”。）

15.2.4.2 自定义类型

想一想，数据结构的定义会产生哪些影响？首先，它引入了新的类型；其次它基于此类型定义若干构造器、谓词和选择器。例如，在上面的例子中，首先引入`BTnum`，然后使用它创建以下类型：

```
BTmt : -> BTnum
BTnd : number * BTnum * BTnum -> BTnum
BTmt? : BTnum -> boolean
BTnd? : BTnum -> boolean
BTnd-n : BTnum -> number
BTnd-l : BTnum -> BTnum
BTnd-r : BTnum -> BTnum
```

观察几个显著的事实：

- 这里的构造器创建 `BTnum` 的实例，而不是更具体的东西。稍后我们将讨论这个设计抉择。
- 这里的谓词函数都接受 `BTnum` 类型参数，而不是“Any”（任意值）。这是因为类型系统已经可以告诉我们某个值的类型是什么，因此我们只需要区分该类型的不同形式。
- 选择器只能作用于类型中相关形式的实例——例如，`BTnd-n` 只对 `BTnd` 的实例有效，对 `BTmt` 的实例则不行——但是由于缺乏合适的静态类型，我们无法在静态类型系统中表示这点。

递归类型中还有很多值得讨论的东西，我们不久将回到这个话题。

15.2.4.3 模式匹配和去语法糖

类型定义的讨论告一段落，剩下要提供的功能就是模式匹配。例如，我们可以这样写：

```
(type-case BTnum t
    [BTnum () e1]
    [BTnd (nv lt rt) e2])
```

我们知道，这可以用前述的函数来实现。用 let 就可以模拟此模式匹配所实现的绑定：

```
(cond
    [(BTmt? t) e1]
    [(BTnd? t) (let ([nv (BTnd-n t)]
                     [lt (BTnd-l t)]
                     [rt (BTnd-r t)]
                 e2)])
```

总之，它可以通过宏实现，所以模式匹配不需要被添加到核心语言中，直接用去语法糖即可实现。这也意味着一门语言可以有很多不同的模式匹配机制。

不过，这不完全正确。生成上面代码中的 `cond` 表达式时，宏需要通过某种手段知道 `BTnd` 的三个位置选择器分别是 `BTnd-n` 、 `BTnd-l` 和 `BTnd-r` 。这些信息在类型定义时显式给出，但是在模式匹配时是隐含的（划重点）。因此，这些信息必须要从类型定义处传过来。因此宏展开器需要使用类似类型环境的东西完成其任务。

此外，还要注意，例如 `e1` 和 `e2` 这样的表达式无法类型检查——事实上，甚至不能被可靠地识别为表达式——直到完成了 `type-case` 的宏展开之后。因此，展开依赖于类型环境，而类型检查依赖于展开的结果。换句话说这两者是共生关系，不仅仅是并行运行，而是同步运行。因此，静态类型语言中进行去语法糖操作时，如果语法糖需要对相关类型作出推测，要比动态类型语言中更复杂一些。

15.2.5 类型、时间和空间

明显，类型已经赋予了类型安全语言一些性能优势。因为一些本来需要运行时执行的检查（例如，检查加法的两个参数的确是数）现在是静态执行的。在静态类型语言中，类似`:number`的注解已经回答了关于某个值是否是特定类型这种问题；无需在运行时再去检查。因此，类型级别的谓词以及程序中对它们的使用将会（并且需要）完全消失。

对于开发者来说这需要付出一些代价，他们必须说服静态类型系统他们的程序不会导致类型错误；由于可判定性的限制，有些可以正确运行的程序也可能与类型系统冲突。不过，类型系统为满足了它要求的程序提供了可观的运行时性能优势。

接下来我们来讨论空间。到目前为止，语言的运行时系统需要对每个值附加存储其类型信息。这也是其实现类型级别谓词如 `number?` 的基础，这些谓词既可被开发人员使用也可被语言内部使用。如果不需要这些谓词，那么这些为了实现它们而存储的信息所占据的空间也将不再需要。因此（静态语言）不需要类型标签。

> 然而，垃圾回收器仍然需要它们，但其他表示法（如BIBOP(译注BIg Bag Of Pages)）能极大减少它们对空间的需求。

类型变体相关的谓词仍要保留：如上面例子中的`BTmt?`和`BTnd?`。它们的调用需要在运行时求值。例如，如前所述，选择器`BTnd-n`就需要执行这种检查。当然，进一步的优化是可能的。考虑模式匹配去语法糖后生成的代码：其中的三个选择器就无需执行这些检查，因为只有`BTnd?`返回真值时才会执行对应代码片。因此，运行时系统可以给去语法糖层面提供特殊的不安全（unsafe）指令，也就是不执行类型检查的版本，从而生成如下所示的代码：

```
(cond
  [(BTmt? t) e1]
  [(BTnd? t) (let ([nv (BTnd-n/no-check t)]
                   [lt (BTnd-l/no-check t)]
                   [rt (BTnd-r/no-check t)])
               e2)])
```

但最终的结果是，运行时系统仍然需要存储足够的信息来准确回答这些问题。不过，相比于之前需要使用足够的位来区分每种类型及类型变体，现在，由于类型被静态地隔离了，对于没有变体的类型（例如，只有一种类型的字符串），不再需要存储任何变体相关的信息；这意味着运行时系统可以使用所有可用位来存储实际的动态值。

与之相对，如果类型存在变体，运行时系统需要牺牲一些空间用于区分不同变体，不过一个类型中变体的数量显然比所有类型和其变体的数量要小得多。在上面的例子中，`BTnum`只有两个变体，因此运行时系统只需要使用一个比特来记录某个值是`BTnum`的哪个变体。

特别要注意的是，类型体系的隔离可以防止混淆。如果有两种不同的数据类型，每种都有两种变体，在动态类型的世界中，所有这四种变体都需要有不同的表示法；与之相对，在静态类型的世界中，这些表示法可以跨类型重叠，因为静态类型系统会保证一种类型中的变体和另一种类型中的不被混淆。因此，类型系统对于程序的空间（节约表示所需空间）和时间（消除运行时检查）上都有实打实的性能提升。

15.2.6 类型和赋值

我们已经覆盖了核心语言中除赋值之外的大部分基本特性。从某些方面看，类型和赋值之间的相互作用很简单，这是因为在经典环境中，它们根本不相互作用。例如，考虑下面动态类型程序：

```
(let ([x 10])
  (begin
    (set! x 5)
    (set! x "某物")))
```

`x` 的“类型”是什么？它并没有确定的类型，它在一段时间内是数，后来（注意里面蕴含时间意味）是字符串。我们根本无法给它定类型。一般来说，类型检查是种非时间性的活动：它只在程序运行之前执行一次，因此必须独立于程序执行的特定顺序。因此，跟踪贮存中的精确值超出了类型检查程序的能力范围。

上面的例子当然可以简单的静态的被理解，不过我们不能被简单的例子误导。考虑下面的程序：

```
(let ([x 10])
  (if (even? (read-number "输入数字"))
      (set! x 5)
      (set! x "某物")))
```

现在，静态检查不可能得到关于 `x` 的类型的结论，因为只有在运行时我们才能获得用户输入的值。

为了避免这种情况，传统的类型检查器采用了一个简单策略：赋值过程中类型必须保持不变。也就是说，赋值操作，不论是变量赋值还是结构体赋值，都不能改变被赋值的量的类型。因此，上面的代码在我们当前的语言中将不能通过类型检查。给程序员提供多少灵活性就取决与语言了。例如，如果我们引入更加灵活的类型表示“数或字符串”，上面的例子将能通过类型检查，但是 `x` 的类型就永远不那么精确，所有使用 `x` 的地方都需要处理这种降低了的精度，后面我们会回到这个问题。

简而言之，在传统的类型系统中赋值相对容易处理，因为它采用了简单的规则，值可以在类型系统指定的限度下进行改变，但是类型不能被改变。在像 `set!` 这种操作的情况下（或者我们的核心语言中的 `setC` ），这意味着赋值的类型必须和变量的类型匹配。在结构体赋值的情况下，例如 `box` ，这意味着赋值的类型必须和 `box` 容器内容的类型匹配。

15.2.7 中心定理：类型的可靠性

之前我们说过，一些静态类型语言可以为其书写的程序所能达成某些特性作出很坚实的证明：例如，该语言书写的程序肯定会终止。当然，一般来说，我们无法获得这样的保证（事实上，正是为了能写出无限循环我们才添加的通用递归）。然而，一个有意义的类型系统——事实上，任何值得类型系统这一高贵头衔的东西【注释】——应该为所有静态类型程序

提供某种有意义的保证。这是给程序员的回报：通过给程序加上类型，她可以确保某些不好的事情不会发生。没有类型的话，我们也能找到bug；这是有用的，但它不足以提供构建高级别工具（例如要保证安全性、隐私性或健壮性）的必要基础。

> 我们一再使用“类型系统”这个术语。类型系统通常是三个组件的组合：类型的语言、类型规则，以及将这些规则应用于程序的算法。我们的讨论中将类型规则放入函数中，因此模糊了第二者和第三者之间的区别，但它们仍然可以在逻辑上加以区分。

我们可能希望类型系统给我们提供什么样的保证呢？请记住，类型检查器在程序运行前静态地对程序进行检查。这意味着它本质上是对程序行为的预测：例如，当它指出某个复杂表达式的类型为 `num`，它实际是在预测程序运行时，该表达式将产生一个数值。我们怎么知道这个预测是正确的呢，也就是说检查器从不撒谎？每种类型系统都应该附带一个证明这一点的定理。

对于类型系统存疑有一个很好的理由，不是怀疑主义的那种。类型检查器和程序求值器工作方式上有很多不同：

- 类型检查器能见到的只有程序文本，求值器运行在真实的存储器上。
- 类型环境将标识符绑定到类型，求值器的环境则绑定标识符到值或者存储位置。
- 类型检查器将值的集合（甚至是无限集合）压缩成类型，而求值器处理的是值本身。
- 类型检查器一定会终止，求值器不一定会。
- 类型检查器仅需检查表达式一遍，求值器运行时某个表达式的运行次数可能从零次到无穷次。

因此，我们不应假设这两者将始终对应！

对于给定的类型系统，我们希望达到的核心目标是——该类型系统是可靠的（sound）。它的意思是：给定表达式（或者程序） `e` ，类型检查得出其类型为 `t` ，当我们运行 `e` 时，假设得到了值 `v` ，那么 `v` 的类型是 `t` 。

证明这个定理的标准方法是分两步进行，进展（progress）和保持（preservation）。进展的意思是，如果一个表达式能够通过类型检查，那么它应该能进行进一步求值得到新的东西（除非它本身就是值）；保持的意思是，这个求值步骤前后类型不变。如果我们交错进行这些步骤（先进展再保持，不断重复），可以得出一个结论，最终的结果和最初被求值的表达式类型相同，因此类型系统确实是可靠的。

例如，考虑表达式： `(+ 5 (* 2 3))` 。它的类型为 `num` 。在一个可靠的类型系统中，进展证明，由于该表达式能通过类型检查，且其当前不是值，它可以进行一步求值——这里它显然可以。进行一步求值之后，它被规约成了 `(+ 5 6)` 。不出所料，正如保持给出的证明，它的类型也为 `num` 。进展表明它还能进行一步求值，得到 `11` 。保持再次表明它的类型和上一步的表达式类型相同，都为 `num` 。现在，进展发现我们已经得到最终结果，无后续要进行的求值步骤，该值的类型和最初的表达式类型相同。

但这不是完整的故事。有两点需要说明：

1. 程序可能不会得出最终的结果，它可能永远循环。这种情况下，该定理严格来说并不适用。但是我们仍能看到，计算得到的中间表达式类型将一直保持不变，因此即使程序没有最终产生一个值，它仍在进行着有意义的计算。
2. 任何特性足够丰富的语言中都存在一些不能静态决定的属性（有些属性也许本来可以，但是语言的设计者决定将其推迟到运行时决定）。当这类属性出错时——比如，数组的索引越界——关于这种程序没有很好的类型可以约束它们。因此，每个类型完备性定理中都隐含了一组已发布的、允许的异常或者可能发生的错误条件。使用该类型系统的开发者隐式的接受了这些条件。

作为第二点的一个例子，典型的静态类型语言中，都会指明对于向量的寻址、链表的索引等操作可能抛出异常。

后面这个说明好像站不住脚。事实上，我们很容易忘记这其实是一条关于运行时不能发生的事情的陈述：这一组异常之外的异常将能被证明不会产生。当然，对最开始就设计为静态类型的语言，除了不那么严格的类比外，可能搞不清这组异常具体是什么，因为一开始本就无须定义它们。但是当我们将类型系统添加到已有的语言时——特别是动态类型语言，如Racket或Python——那么这里已经有一组明确定义的异常，类型检查器将会指明其中一些异常（像“函数调用位置不是函数”或者“未找到方法”）不会发生。这就是程序员接纳类型系统语法上限制所得到的回报。

15.3 对核心的扩展

现在我们已经有了基础的静态类型语言，下面探索一下如何将其扩展成为更有用的编程语言。

15.3.1 显式的参数多态

下面哪些是相同的？

- `List<String>`
- `List<String>`
- `(listof string)`

事实上，上面任何两个都不太一样。但是第一个和第三个非常相似，因为第一个是Java代码而第三个是我们的静态语言代码，而第二个，是C++代码，和其它两个不同。清楚了吗？不清楚？很好，继续往下读！

15.3.1.1 参数化类型

我们所使用的编程语言已经展示了参数多态的价值，例如，`map` 函数的类型可以这样给出：

```
(('a -> 'b) (listof 'a) -> (listof 'b))
```

意思是，对于任意类型 `'a` 和 `'b` ， `map` 读入一个从 `'a` 到 `'b` 的函数，一个 `'a` 的链表，生成对应的 `'b` 的链表。这里， `'a` 和 `'b` 不是具体的类型；它们是类型变量（我们的术语中，这应该被称为“类型标识符”，因为它们在实例化过程中不会变化；但是我们还是使用传统术语）。

可以换种方式理解它：实际上有一族无穷多的这样的 `map` 函数。例如，其中一个 `map` 的类型是这样的：

```
((number -> string) (listof number) -> (listof string))
```

另一个的类型是这样的（没有限制说其中的类型必须是基本类型）：

```
((number -> (number -> number)) (listof number) -> (listof (number -> number)))
```

还有这样的（也没有限制说 `'a` 和 `'b` 必须不同）：

```
((string -> string) (listof string) -> (listof string))
```

以此类推。由于它们的类型不同，名字也需要不同： `map_num_str` 、 `map_num_num->num` 、 `map_str_str` 等。但是这会让它们变成不同的函数，于是我们总得使用某个特定 `map` ，而不是直接使用比较一般的那个。

显然，不可能将所有这些函数放到我们的标准库中：毕竟它们有无穷多个！更好的方式是能按需获取我们需要的函数。我们的命名规则给出了一点提示： `map` 接受两个参数，它们都是类型。给定了两个类型作为参数，我们可以得到针对特定类型的 `map` 函数。这种类型的参数化被称为参数多态。

> 注意不要和对象“多态”搞混，后面会讨论它。

15.3.1.2 显式声明类型参数

换句话说，我们相当于说 `map` 实际上是有四个参数的函数，其中两个是类型，另外两个是实际的值（函数和链表）。在需要显式声明类型的语言中，我们需要写成类似这样：

```
(define (map [a : ???] [b : ???] [f : (a -> b)] [l : (listof a)]) : (listof b)
  ...)
```

但是这会产生一些问题。首先， `???` 处应该填什么？它是 `a` 和 `b` 的类型。但是如果a和b本身将被类型替换，那么类型的类型是什么？其次，我们真的希望每次调用map的时候传入四个参数吗？再者，我们真的希望在接收任何实际值之前先接收类型参数吗？对于这些问题的答案能延伸出关于多态类型系统巨大的讨论空间，其中的大部分我们这里将不会涉及。

> 推荐阅读Pierce的《Types and Programming Languages(类型和编程语言)》，获取易懂、现代的介绍。

注意到一旦我们引入参数化，很多预期之外的代码都将被参数化。例如，考虑平平无奇的 `cons` 函数的类型。它的类型需要基于链表中值的类型进行参数化（尽管它实际上并不依赖于这些值——稍后会解释这一点），于是每次使用 `cons` 时都需要正确地进行类型实例化。说到这，即使用 `empty` 创建空链表也必须类型实例化！当然，Java和C++程序员应该对这个痛点很熟悉了。

15.3.1.3 一阶多态

我们将只讨论这个空间中一个特别有用且易于理解的点上，也即 Standard ML 的类型系统、同时是本书使用的静态类型语言和早期版本的 Haskell 的类型系统，有范型加成的 Java 和 C# 以及引入了模版的C++ 也差不多获得了这种类型系统的大部分能力。这类语言定义了被称为谓词、一阶或者叫前缀多态的东西。关于上小节的问题它的答案是不填、没有、是。下面我们来探讨一下。

我们首先将类型的世界分成两组。第一组包含我们目前用到的静态类型语言，另外加上类型变量；它们被称为 **monotype**（单型）。第二组包含参数化的类型，被称为 **polytype**（多型）；按惯例它们是这样写的：`∀` 前缀，一组类型变量，再跟一个类型表达式，表达式中可以使用这些类型变量。因此，`map` 的类型将写作：

```
∀ a, b : (('a -> 'b) (listof 'a) -> (listof 'b))
```

由于“`∀`”是逻辑符号“对于所有的”的意思，于是上面的东西可以读作：“对于所有类型 `'a` 和 `'b`，`map` 的类型为……”。

在一阶多态（rank-1 polymorphism）中，类型变量只能被 `monotype` 替换。（此外，它们只能被具体类型替换，否则剩下的类型变量将无法被替换掉。）因此，在类型变量参数和常规参数之间我们有了明确的界线。我们不需要为类型变量提供“类型注解”，因为我们知道它们可以是什么。这样得到的语言相对简洁，但仍提供了相当的表达能力。

> 非直谓性语言（Impredicative language）取消了 `monotype` 和 `polytype` 的区别，因此类型变量可以使用另一个多态类型实例化。

注意到由于类型变量只能被 `monotype` 替换，他们全相互对立。于是，类型参数可以全被提到参数表的前面。这使我们可以使用形如 `∀ tv, ... : t` 的类型，其中 `tv` 是类型变量，`t` 是 `monotype`（其中可以引用这些类型变量）。此语法的意义就在这里，这也是之前称其为前缀多态的原因。而且后面也将看到这对其实现也很有用。

15.3.1.4 通过去语法糖实现一阶多态解释器

该特性最简单的实现就是将其视为一种去语法糖的形式：C++ 实际上就是这么做的。（具体来说，因为 C++ 有一个叫做模版的宏系统，所以使用模版，它非常巧合地达成了一阶多态。）举个例子，如果我们有一个语法形式 `define-poly`，它接收名字、类型变量和表达式。当传入类型的时候，它将表达式中对应类型变量替换为此类型，因此：

```
(define-poly (id t) (lambda ([x : t]) : t x))
```

通过将 `id` 定义为多态的方式定义了一个恒等（identity）函数：给 `t` 传入递任意具体类型，就得到一个单参数的类型为 `(t -> t)` 的函数（其中 `t` 被替换）。我们可以使用各种类型实例化 `id` ：

```
(define id_num (id number))
(define id_str (id string))
```

从而获得针对这些类型的恒等函数：

```
(test (id_num 5) 5)
(test (id_str "x")  "x")
```

与之相对，像

```
(id_num "x")
(id_str 5)
```

这样的表达式将不能通过类型检查（而不是运行时出错）。

如果你好奇的话，下面给出了实现。简单起见，我们假设只有一个类型参数；很容易使用 `...` 实现多个参数的情形。我们不仅将 `define-poly` 定义为宏，它还会定义宏：

```
(define-syntax define-poly
  (syntax-rules ()
    [(_ (name tyvar) body)
     (define-syntax (name stx)
       (syntax-case stx ()
         [(_ type)
          (with-syntax ([tyvar #'type])
            #'body)]))]))
```

因此，对于：

```
(define-poly (id t) (lambda ([x : t]) : t x))
```

该语言将创建名为 `id` 的宏：对应 `(define-syntax (name ...) ...)` 的部分（对于这个例子， `name` 是 `id` ）。 `id` 的一个实例，如 `(id number)` ，将类型变量 `t` 、宏里面的 `typvar` 替换成给定的类型。因为要规避卫生，我们用 `with-syntax` 来确保所有对于类型变量（typvar）的使用被替换为给定的类型。因此，实际效果是，

```
(define id_num (id number))
```

被转换成了

```
(define id_num (lambda ([x : number]) : number x))
```

然而这种方式有两个重大局限性：

1. 来试试定义递归的多态函数，比如说 `filter` 。之前我们说过，每个多态值（例如 `cons` 和 `empty` ）都需要类型实例化，但是为了简洁起见我们将依赖静态类型语言实现这点，而仅专注于 `filter` 的类型参数。对应代码是：

   ```
   (define-poly (filter t)
     (lambda ([f : (t -> boolean)] [l : (listof t)]) : (listof t)
       (cond
         [(empty? l) empty]
         [(cons? l) (if (f (first l))
                        (cons (first l)
                              ((filter t) f (rest l)))
                        ((filter t) f (rest l)))])))
   ```

   注意到递归的使用 `filter` 时，必须使用恰当的类型对其实例化。

   上面的定义完全正确，只有一个问题，当我们尝试使用它时——如：

   ```
   (define filter_num (filter number))
   ```

   DrRacket 将不会终止，更准确的说，是宏展开不会终止，因为它将不断的尝试创建 `filter` 代码的副本。不过如果用下面这种方式定义该函数，展开会终止——

   ```
   (define-poly (filter2 t)
     (letrec ([fltr
               (lambda ([f : (t -> boolean)] [l : (listof t)]) : (listof t)
                 (cond
                   [(empty? l) empty]
                   [(cons? l) (if (f (first l))
                                  (cons (first l) (fltr f (rest l)))
                                  (fltr f (rest l)))]))])
       fltr))
   ```

   但是这给开发人员徒增了不必要的痛苦。实际上，一些模版展开程序会缓存之前展开的值，避免对于相同的参数反复生成代码。（Racket 做不到这点，因为一般来说，宏表达式可以依赖可变变量和值，甚至可以执行输入输出，因此 Racket 无法保证同样的输入表达式总是产生相同输出。）

2. 考虑恒等函数的两个实例。我们无法比较 `id_num` 和 `id_str` ，因为它们类型不同，但即使它们类型相同，使用 `eq?` 比较它们也不同：

   ```
   (test (eq? (id number) (id number)) #f)
   ```

> 这是因为对 `id` 每次实例化都会创建一份新的代码副本。即使使用了上面提到的优化，同一种类型对应代码只有一份副本，但是不同类型的对应代码体还是会被重新生成【注释】——但这也是没必要的！例如， `id` 的实现的部分其实没任何东西依赖于参数的类型。实际上， `id` 这一族无穷多个的函数可以共享同一个实现。简单的去语法糖策略实现不了这点。
>
> > 事实上， `C++` 模版因代码膨胀的问题而臭名昭著，这是原因之一。

换种说法，基于去语法糖的策略本质上是使用替换的实现方式，它有着和我们之前函数调用时使用替换的方式实现相同的问题。不过，其它情况下，替换策略能达成我们关于程序行为的期望；对于多态也是一样，正如我们将看到的一样。

注意去语法糖策略的一个好处就是它不需要类型检查器“理解”多态。我们的核心语言仍可以是单态的（monomorphic），所有的（一阶）多态完全由宏展开处理。这提供了一种廉价的将多态添加到语言中的策略，但正如C++所示，它也引入了很大的开销。

最后，虽然这里我们只关注了函数，但前面的讨论同样适用于数据结构。

15.3.1.5 其它实现方式

有些其他实现策略不会遇到此类问题。这里我们不会深入讲解它们，但是其中一些策略的本质就是上面提到过的“缓存”方法。因为可以确定的是，对于给定的同一组类型参数，应该得到相同的实现代码，不需要对相同的类型参数实例化多次。这避免了无限循环。如果我们检查了使用特定类型实例化的代码一次，后续相同类型参数的实例化结果就无需再进行类型检查（因为它不会发生改变）。此外，我们无需保留实例化后的源码：一旦我们检查了展开后的程序，就可以将其丢弃，运行时也只需要保留一份实例化的副本。这样可以避免上述纯去语法糖策略中讨论过的所有问题，同时保留它的好处。

其实我们有点过分了。静态类型的好处之一就是能选择更精确的运行时表示。例如，静态类型可以告诉我们用的是数是32位的还是64位的甚至1位的（也就是布尔值）。然后编译器可以利用位的布局方式（例如，32个布尔值可以*打包进一个32位字）为每种表示生成专用代码。因此，在对每种使用的类型进行检查之后，多态实例化程序可以跟踪函数或数据结构使用时用到的特定类型，并将这些信息提供给编译器用于代码生成。这会导致生成相关函数的若干副本，彼此都互不 `eq?` ——但这么做有充分的理由，因为它们要执行的操作的确不同，所以这是正确的。

15.3.1.6 关系型参数

我们还需解决关于多态的最后一个细节。

早先我们说过像 `cons` 这样的函数不依赖于其参数的具体值。这一点对 `map` 、 `filter` 等也成立。 `map` 和 `filter` 接收一个函数作为参数，当它们要对单个元素进行操作时，实际上使用该函数进行操作，即该函数负责做出如何处理元素的决定； `map` 和 `filter` 本身只是遵从该函数参数。

“检验”这种情况是否属实的一种方法是，替换不同类型的值链表及对应的函数作为参数。也就是说假设两组值之间有映射关系；我们根据此关系替换链表元素和参数函数。问题是，`map` 和 `filter` 的输出结果是否可以通过该关系预测？如果对于某些输入，`map` 的输出和关系预测的结果不同，这说明 `map` 肯定侦测了实际值并根据相关信息做出了处理。但事实上，这不会发生在 `map` 上，或者说实际上也不会发生在大多标准多态函数上。

遵从这类型关系准则的函数被称为关系型参数（Relational Parametricity）【注释】。这是类型赋予我们的另一个非常强大的能力，因为它们告诉我们这种多态函数可以执行的操作很受限制：它们可以删除、复制或重新排列元素，但是不能考察这些元素，也不能对它们进行具体操纵。

> 请参阅Wadler的《Theorems for Free!》和Reynolds的《Types, Abstraction and Parametric Polymorphism》。

起初这听起来非常令人印象深刻（确实如此！），但细查，你可能会意识到这与经验并不一致。例如，在Java中，多态方法依然可以使用 `instanceof` 在运行时检查、获得特定类型的值，并相应的改变行为。这种方法就不是关系型参数了！【注释】事实上，关系型参数也能被看作是语言弱点的一种表述：它只允许一组有限的操作。（你仍可以检查类型——但不能根据你获取的信息进行相关行动，这样检查就没有意义了。因此运行时系统如果想要模拟关系型参数，必须要移除类似 `instanceof` 及它的替代行为：例如，对值进行加一操作并捕获异常以判断它是数。）然而，这是个非常优雅和令人吃惊的结果，显示了使用丰富类型系统能获得的强大程序推理能力。

> 网上，你会经常发现这个属性被描述为函数不能检查其参数——这是不正确的。

15.3.2 类型推断

手工书写每处多态类型的实例参数是一个令人沮丧的过程，很多版本的Java和C++用户可以证明这点。想象一下，每次使用 `first` 和 `rest` 时都需要传入类型参数是个什么场景！我们之所以能够避免这种命运，是因为我们的语言实现了类型推断。这使我们可以编写定义：

```
(define (mapper f l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (f (first l)) (mapper f (rest l)))]))
```

然后编程环境自动声明

```
> mapper
- (('a -> 'b) (listof 'a) -> (listof 'b))
```

它不仅是正确的类型，而且是非常一般的类型！从程序结构中派生出这种一般类型的过程感觉几乎就是魔法。我们来揭示其幕后。

首先，我们来了解类型推断做了什么。有些人错误的认为，有类型推断的语言无类型声明，其被类型推断取而代之了。这混淆了多个层面的东西。首先，即使在有类型推断的语言中，程序员仍被允许声明类型（并且为了文档更为清晰，通常会鼓励这样做——就像你之前被鼓励的一样）【注释】。此外，在没有这些声明的情况下，推断的实际含义并不显明。

> 有时（类型）推断是不可判定的，这时程序员别无选择只能声明某些类型。最后，显式的书写类型注解能够大大减少难以辨认的错误信息。

相反，最好将底层语言看作需要完整地显式声明类型的——就如我们刚才研究的多态语言。然后我们说，在`:`后类型注解部分可以留空，编程环境中的某个特性会为我们填充这些。（如果走得更远，我们可以丢弃`:`及额外的修饰，它们都会被自动插入。因此，类型推断只是为用户提供的一种便利，减轻编写类型注解的负担，而底层的语言仍然是显式声明类型的。

我们怎么考虑类型推断做的是什么呢？假设我们有个表达式（或者程序）`e`，由显式声明类型语言书写：也就是说在任何需要类型注解的地方都有写出。现在假设我们擦除`e`中所有的类型注解，然后使用函数`infer`将它们推断回来。

思考题

> `infer`应该有何种属性？

我们可以要求很多东西。其中之一为，它要产生和`e`原来恰好一样的注解。这在很多方面都是有问题的，尤其是当`e`本就不能通过类型检查的情况下，怎么能推断回它们（应该）是什么？你可能觉得这是个学究式的玩笑：毕竟，如果`e`本就不能通过类型检查，如果能在删除其注解之后还能还原回来呢？反正两者都不能通过类型检查，谁在乎啊？

思考题

> 这个推理正确吗？

假设`e`是：

```
(lambda ([x : number]) : string x)
```

它显然不能通过类型检查。但是如果我们擦除类型注解——得到

```
(lambda (x) x)
```

——这个函数显然可以合法地添加类型！因此，更合理的需求可以是，如果原始的`e`能通过类型检查，那么对应的使用了推导出的注解的版本也必须能。这种单向的含义的用途体现在两方面：

1. 它没有说`e`未通过类型检查应该怎样，也即它不会排除前述的类型推断算法，其会将例子中类型错误的恒等函数变成类型正确的。

2. 更重要的是，它向我们保证，使用类型推断将不会使我们失去任何东西：之前能通过类型检测的程序不会被推断后而不能。这意味着我们可以在想要的地方显式添加类型注解，但不会被迫这样做。

> 当然，这只在程序推断可判定的情况下才成立。

我们还可能希望两者类型是相同的，但这不是能做到的：函数

```
(lambda ([x : number]) : number x)
```

类型为 `(number -> number)` ，而擦除类型注解后推导出的类型要一般得多。因此，将这些类型关联并给出类型相等的定义并不简单，尽管如此后面将简要讨论此问题。

有了这些准备，我们下面进入对类型推断机制的研究。最需要注意的地方，前述的简单递归下降的类型检查算法将不再起作用。它之前能起作用，是因为所有函数的边界处都有类型注解，所以我们下降进入函数体，同时用类型环境携带这些注解中包含的信息。没了这些注解，就不知如何递归下降了。

事实上，目前还不清楚哪个方向更合理。像上面 `mapper` 的定义，各代码段之间互相影响。例如，从 `empty?` 、 `cons?` 、 `first` 和 `rest` 对 `l` 的调用都可以看出它是链表。但是是什么的链表呢？从这些操作看不出来。然而，对于其每个（或者应该说，任意） `first` 元素调用了 `f` 这点可以看出，链表成员的类型必须可以被传给 `f` 。同理，由 `empty` 和 `cons` 我们可以知道（ `mapper` 的）返回表达式必须为链表。它的成员类型是什么呢？必须为 `f` 的返回类型。最后，请注意最微妙的地方：当参数链表为空时，我们返回 `empty` 而不是 `l` （这时我们是知道其被绑定到 `empty` ）。使用前者，返回值的类型可能是任意类型的链表（仅受 `f` 返回类型的约束）；使用后者，返回的类型就被迫和参数链表的类型相同。

所有这些信息都包含在函数里。但是我们如何系统地提取出这些信息呢，而且使用的算法必须会终止，并满足前面陈述属性？我们分两步来做。首先，根据程序表达式生成其必须要满足的类型约束。然后，通过合并散布在函数体各处的约束、识别其中的不一致，最终解决约束。每一步都相对简单，但是组合起来创造了魔力。

15.3.2.1 约束生成

我们最终的目标是给每个类型注解位置填入类型。将会证明，这也等同于找到每个表达式的类型。简单想想就知道，这本来也是必要的：比如，在不知道函数体类型的情况下，如何能确定函数本身的类型？这也是足够的，因为如果每个表达式的类型都被计算得出，其中必然包括了那些需要被注解的表达式。

首先，我们需要生成（待解决的）约束。这一步会遍历程序源码，为每个表达式生成恰当的约束，最后返回这组约束。为了简单，使用递归下降的方式实现；它最终生成约束的集合，所以原则上遍历和生成的顺序是无关紧要的——因此我们选择了相对简单的递归下降方式——当然，为了简单起见，我们使用链表表示这个集合。

约束是什么呢？就是关于表达式类型的陈述。此外，虽然变量绑定并不是表达式，但我们仍需计算其类型（因为函数需要参数和返回值类型）。一般来说，对于表达式的类型我们知道些什么呢？

1. 它和某些标识符的类型有关。
2. 它和某些其它表达式的类型有关。
3. 它是数。
4. 它是函数，其定义域（domain）和值域（range）类型可能受到进一步的约束。

因此，我们定义如下两个数据结构：

```
(define-type Constraints
  [eqCon (lhs : Term) (rhs : Term)])

(define-type Term
  [tExp (e : ExprC)]
  [tVar (s : symbol)]
  [tNum]
  [tArrow (dom : Term) (rng : Term)])
```

接下来定义约束生成函数：

```
<constr-gen> ::= ;约束生成

    (define (cg [e : ExprC]) : (listof Constraints)
      (type-case ExprC e
        <constr-gen-numC-case>
        <constr-gen-idC-case>
        <constr-gen-plusC/multC-case>
        <constr-gen-appC-case>
        <constr-gen-lamC-case>))
```

当表达式为数时，唯一能说的是，我们希望该表达式的类型为数类型：

```
<constr-gen-numC-case> ::=

    [numC (_) (list (eqCon (tExp e) (tNum)))]
```

听上去很微不足道，但我们不知道的是，其他包含它的表达式是什么。因此，某个更大的表达式可能会与此断言——这个表达式的类型必须是数型——相矛盾，从而导致类型错误。

对于标识符，我们只是简单地说，表达式的类型就是我们所期望该标识符应有的类型：

```
<constr-gen-idC-case> ::=

    [idC (s) (list (eqCon (tExp e) (tVar s)))]
```

如果上下文限制了其类型，该表达式的类型将自动受到限制，并且必须与上下文的期望一致。

加法是我们第一个遇到的上下文约束。对于加法表达式，首先需要确保我们生成（并返回）其两个子表达式的约束，而子表达式可以是复杂的。这两个约束中，我们期望什么？需要每个子表达式是数类型的。（如果其中一个子表达式不是数类型的，应该导致类型错误。）最后，我们断言整个表达式的类型为数。

```
<constr-gen-plusC/multC-case> ::=

    [plusC (l r) (append3 (cg l)
                          (cg r)
                          (list (eqCon (tExp l) (tNum))
                                (eqCon (tExp r) (tNum))
                                (eqCon (tExp e) (tNum)))))]
```

> `append3` 是 `append` 的三参数版本。

`multC` 的情况与之相同，区别只在名字上。

下面我们来看另外两个有趣的情况，函数声明和调用。两种情况下我们都需要生成和返回子表达式的约束。

在函数定义中，函数的类型是函数（“箭头/arrow”）类型，其参数类型是形参的类型，其返回类型是函数体的类型。

```
<constr-gen-lamC-case> ::=

    [lamC (a b) (append (cg b)
                        (list (eqCon (tExp e) (tArrow (tVar a) (tExp b)))))]
```

最终，考虑函数调用。我们不能直接陈述函数调用的类型约束。不过，我们可以说，函数接受的参数类型必须和实际参数的类型相同，并且函数返回的类型就是调用表达式的类型。

```
<constr-gen-appC-case> ::=

    [appC (f a) (append3 (cg f)
                         (cg a)
                         (list (eqCon (tExp f) (tArrow (tExp a) (tExp e)))))]
```

完成了！我们已经完成约束的生成；现在只需解出它们。

15.3.2.2 使用合一求解约束

求解约束的过程也被称为合一（unification）。合一器的输入是等式的集合，其中每个等式是变量到项（term）的映射，项的数据类型在上面定义了。注意到一点，我们实际上有两种变量。`tvar` 和 `tExp` 都是“变量”，前者很明显，注意后者同样也是，因为我们需要求解此类表达式的类型。（另一种方式是为每个表达式引入新的类型变量，但我们仍需一种方法确定这些变量与表达式之间的对应关系，而现在这已经能通过对表达式进行 `eq?` 操作自动完成了。另外这会产生大得多的约束集，不好进行人工检查。）

就我们的目的而言，合一是为了是生成替换（substitution），或者说将变量映射为不包含任何变量的项。这听起来应该很耳熟：我们有一组联立方程，其中每个变量都是线性使用的；这种方程组可以使用高斯消元法求解。该情形中，我们清楚最终可能遇到缺少约束（under-constrained）或过度约束（over-constrained）的情况。这种事情同样也将发生这里。

合一算法会遍历约束集合。由于每个约束有两项，每个项有四种可能的类型，因此有十六种情况需要考虑。幸运的是，我们实际可以用比较少的代码覆盖这十六种情况。

算法从所有约束的集合和空替换开始。每个约束都会被处理一次，并从集合中删除，因此原则上终止判据应该非常简单，但是实际处理起来还有点小麻烦。随着约束被处理，替换集合会逐渐增长。当所有的约束都被处理完后，合一过程返回最后的替换集合。

对于给定的约束，合一器检查等式左边，如果它是变量，那么这时它就可以被消除了，合一器将该变量（等式）的右侧添加到替换中，为了眞正完成消除，还需要将替换集中所有该变量的出现替换成该右侧。实践中，实现需要考虑效率；例如，使用可变值表示这些变量可以避免搜索—替换过程。然而我们可能需要进行回溯（我们在后面确实会需要），可变值表示也有缺点。

思考题

> 注意到上面微妙的错误了吗？

这个微妙的错误是，我们说合一器通过替换变量的所有实例来消除它。不过，我们假设等式右侧不包含该变量的实例。不然的话，我们将得到循环定义，这将使替换变得不可能。出于这个原因，合一器会进行出现检查（occurs check）：检查某个变量是否出现在等式两侧，如果是，则拒绝合一。

思考题

> 构造一个其约束会触发出现检查的项。

还记得 `ω` 吗？

下面考虑合一的实现。惯例使用希腊字母 `Θ` 表示替换。

```
(define-type-alias Subst (listof Substitution))
(define-type Substitution
  [sub [var : Term] [is : Term]])

(define (unify [cs : (listof Constraints)]) : Subst
  (unify/Θ cs empty))
```

首先把简单的东西写出来：

```
<unify/0> ::=

    (define (unify/0 [cs : (listof Constraints)] [0 : Subst]) : Subst
      (cond
        [(empty? cs) 0]
        [(cons? cs)
         (let ([l (eqCon-lhs (first cs))]
               [r (eqCon-rhs (first cs))])
           (type-case Term l
             <unify/0-tVar-case>
             <unify/0-tExp-case>
             <unify/0-tNum-case>
             <unify/0-tArrow-case>))]))
```

现在可以实现合一的核心了。我们需要一个辅助函数`extend-replace`，其签名为`(Term Term Subst -> Subst)`。它将执行出现检查，如果检查得出没有环路，则扩展替换集合，并将替换集合中所有出现的第一个项（第一个参数）替代为第二个项（第二个参数）。同样，我们假设`lookup: (Term subst -> (optionof Term))`存在。

练习题

> 定义`extend-replace`和`lookup`。

如果约束等式的左侧是个变量，我们先在替换集合中寻找它。如果存在，我们将当前约束换成新的约束；否则我们扩展替换集合。

```
<unify/0-tVar-case> ::=

    [tVar (s) (type-case (optionof Term) (lookup l 0)
                [some (bound)
                      (unify/0 (cons (eqCon bound r)
                                     (rest cs))
                               0)]
                [none ()
                      (unify/0 (rest cs)
                               (extend+replace l r 0))])]
```

同样的逻辑也适用于表达式的情况：

```
<unify/0-tExp-case> ::=

    [tExp (e) (type-case (optionof Term) (lookup l 0)
                [some (bound)
                      (unify/0 (cons (eqCon bound r)
                                     (rest cs))
                               0)]
                [none ()
                      (unify/0 (rest cs)
                               (extend+replace l r 0))])]
```

如果是基本类型，例如数，我们就需要检查等式右边。有四种可能：

- 如果是数，那么该等式声明类型`num`等于`num`，这恒为真。因此我们可以忽略该约束——它没有告诉我们什么有用信息——继续检查剩下的。 当然，首先得解释为什么会出现这种约束。显然，我们的约束生成器不会生成这种约束。然而，前面替换集合的扩展

会导致这种情况。事实是实践中我们会遇到好几个这种情况。

- 如果是函数类型，显然存在类型错误，因为数和函数类型不相交。同样，我们不会直接生成这样的约束，一定是由先前的替代产生。
- 它可能是两种变量类型之一。不过，我们的约束生成器经过了仔细的安排，不会将它们放在右侧。此外，替代过程也不会在右侧引入它们。因此，这两种情况不会发生。

于是得出这样的代码：

```
<unify/0-tNum-case> ::=

    [tNum () (type-case Term r
               [tNum () (unify/0 (rest cs) 0)]
               [else (error 'unify "number and something else")])]
```

最后还剩下函数类型。这里的论点几乎和数类型完全一样。

```
<unify/0-tArrow-case> ::=

    [tArrow (d r) (type-case Term r
                    [tArrow (d2 r2)
                            (unify/0 (cons (eqCon d d2)
                                           (cons (eqCon r r2)
                                                 cs))
                                     0)]
                    [else (error 'unify "arrow and something else")])]
```

请注意，我们并没有严格地缩小约束集合，因此仅通过约束集合的大小不足以判断这个过程会终止。需要同时综合考虑约束集合的大小以及替换的大小（包括其中变量的个数）。

上面的算法非常通用，不仅对数和函数，对于各种类型项也都适用。我们使用数代表各种基础类型；同样，使用函数代表各种构造类型，例如 `listof` 和 `vectorof` 。

这就完成了。合一产生了替换。现在我们可以遍历这些替换，找到程序中所有表达式的类型，然后插入对应的类型注解。有定理（这里不证明）指出，上面过程的成功意味着程序通过了类型检查，因此我们无需对该程序显式地再跑一遍类型检查。

不过请注意，类型错误的性质在这里发生了巨大变化。之前，我们的递归下降算法利用类型环境遍历表达式。类型环境中的绑定是程序员定义的类型，因此可以被当作（期望的）权威的类型规范（specification）。因此，所有的错误都应归咎于表达式，类型错误的报告很简单（而且很好懂）。然而这里，类型错误无法通知。合一错误是两个智能算法——约束生成和合一——共同导致的，因此程序员不一定能理解。特别是，由于约束的本质是等式，报告的错误位置和“真实”的错误位置可能相差甚远。因此，生成更好的错误信息仍然是个活跃的研究领域。

> 实践中，算法会维护涉及到的程序源码的元信息，并可能也会保存合一的历史，以便溯源错误回源程序。

最后，请记住，约束可能不会精确指明所有变量的类型。如果方程组过度约束，可能会有冲突，导致类型错误。如果缺少约束，这意味着我们没有足够的信息对所有表达式做出明确的类型声明。例如，对于表达式 `(lambda (x) x)` ，没有足够的约束指明 `x` 的类型，从而无法以指明整个表达式的类型。这并非错误；它只是意味着 `x` 可以是任意类型。换句话说，该表达式的类型是“ `x` 的类型-> `x` 的类型”，无其它约束。这些欠约束标识符的类型以类型变量的方式展示，于是上面表达式的类型可以表示为 `('a -> 'a)` 。

合一算法实际上有个很好的属性：它能自动计算表达式最通用的类型，也被称为主类型（principal type）。这就是说，表达式可以有的任何实际类型都可以通过（用实际类型）替换推导出的类型中的类型变量的得到。这是个异乎寻常的结果：没人能生成比前述算法得出的更为一般的类型！

15.3.2.3 Let-多态

很不幸，尽管这些类型变量表面上看和我们之前遇到的多态有诸多相似之处，但它们并不同。考虑下面的程序：

```
(let ([id (lambda (x) x)])
  (if (id true)
      (id 5)
      (id 6)))
```

如果加上显式的类型注解，它能通过类型检查：

```
(if ((id boolean) true)
    ((id number) 5)
    ((id number) 6))
```

然而，如果使用类型推断，它将不能通过类型检查！因为 `id` 中的类型 `'a` ——取决于约束处理的顺序——要么和 `boolean` 合一，要么和 `number` 合一。对应的，那时 `id` 的类型要么是 `(boolean -> boolean)` 要么是 `(number -> number)` 。当使用另一个类型调用 `id` 时，就会发生类型错误！

这是因为我们通过合一推断出来的类型实际并不是多态的。这点很重要：将其称为类型变量不会使你获得多态!类型变量可以在下次使用时合一，彼时，最终得到的还只是单态函数。而真正的多态只有在能真正进行类型变量实例化时才会获得。

所以在具有真正多态的语言中，约束生成和合一是不够的。相反，像ML和Haskell这种语言，甚至我们使用的静态类型语言也是，都实现了俗称**let-**多态的东西。这种策略中，当包含类型变量的项在词法环境中被绑定时，该类型被自动提升为量化类型。每次使用时，该项被自动实例化。

很多实现策略可以做到这点。最简单（而不令人满意）的方式只需复制绑定标识符代码的代码；这样，上面每次 `id` 的使用都会得到自己的 `(lambda (x) x)` 副本，所以每个都有它自己的类型变量。第一个的类型可能是 `('a -> 'a)` ，第二个是 `('b -> 'b)` ，第三个

是`('c -> 'c)`，等等。这些类型变量互不冲突，因此我们得到多态的效果。显然，这不仅增加了程序的大小，而且在存在递归的情况下也不起作用。然而，这给我们提供了通往更好解决方案的思路：不是复制代码，而是复制类型。因此在每次使用时，我们创建推导出类型的重命名版本：第一次使用时，id的类型`('a -> 'a)`变成了`('b -> 'b)`，以此类推，这种方式实现了拷贝代码相同的效果且没有它的包袱。不过，因为这些策略实质都是效仿代码拷贝，因此它们只能在词法环境下工作。

15.3.3 联合类型

假设我们要建立动物园动物的链表，动物有这些种类：犰狳、红尾蚺等。目前，我们必须创建新的数据类型：

```
(define-type Animal
  [armadillo (alive? : boolean)] ;犰狳
  [boa (length : number)]) ;蚺
```

> “在德州，马路中间除了黄线和死掉的犰狳什么都没有。”—— Jim Hightower

然后创建它的链表：`(listof Animal)`。因此，`Animal`类型表示的是`armadillo`和`boa`的“联合（或称联合体，union）”，不过要创建这种联合的唯一方式是每次都创建新类型：比如要创建动物和植物的联合，就需要：

```
(define-type LivingThings
  [animal (a : Animal)]
  [plant (p : Plant)])
```

这样实际的动物现在裹在了更深一“层”。这些类型被称为带标签的联合（tagged union）或可辨识的联合（discriminated union），因为我们需要显式引入类似`animal`和`plant`的标签（或称辨识符(discriminator)）来区分它们。相应地，结构体只能通过数据类型声明来定义；要创建只包含一种变体的数据结构，如

```
(define-type Constraints
  [eqCon (lhs : Term) (rhs : Term)])
```

来表示该数据结构，我们需要使用类型`Constraints`而不是`eqCons`，因为`eqCons`不是类型，只是能在运行时区分的类型变体。

无论哪种方式，联合类型的要点是表示析取或“或”。值的类型是联合中某个类型。值通常只能是联合中某个特定的类型，不过这取决于联合类型的精确定义、规范它们的规则等等。

15.3.3.1 作为类型的结构体

对此自然的反应可能是，为什么不移除这种限制？为什么不允许每个结构体独立存在，将类型定义为一些结构体的集合？毕竟，不管是C还是Racket，程序员都可以定义独立的结构体，无需使用标签构造函数将它们包裹在其它类型里！例如，Racket里可以写：

```
(struct armadillo (alive?))
(struct boa (length))
```

加个注释：

```
;; 动物是下面两者之一：
;; - (armadillo <boolean>)
;; - (boa <number>)
```

但是由于Racket不强制静态类型，这种比较不太清楚。然而，我们可以和 *Typed Racket* （内置与DrRacket中的静态类型Racket）相比较。下面是对应的静态类型代码：

```
#lang typed/racket

(struct: armadillo ([alive? : Boolean]))
(struct: boa ([length : Real])) ;; feet
```

无需引用 `armadillo` 就可以定义使用 `boa` 类型值的函数：

```
;; http://en.wikipedia.org/wiki/Boa_constrictor#Size_and_weight
(define: (big-one? [b : boa]) : Boolean
  (> (boa-length b) 8))
```

事实上，如果调用此函数时传入其它类型，如 `armadillo` —— `(big-one? (armadillo true))` ——将发生静态错误。因为 `armadillo` 和 `boa` 之间的关系等同与数和字符串之间的关系。

当然，我们仍可以定义这些类型的联合：

```
(define-type Animal (U armadillo boa))
```

在这之上定义函数：

```
(define: (safe-to-transport? [a : Animal]) : Boolean
  (cond
    [(boa? a) (not (big-one? a))]
    [(armadillo? a) (armadillo-alive? a)]))
```

之前我们有一种包含两个变体的类型，现在则有三种类型，其中两种类型恰巧能方便的通过联合定义第三种。

15.3.3.2 无标签联合

看起来我们好像还需要辨识标签，但并非如此。在支持联合类型的语言中，通常这样获取类型构造器`optionof`：将期望的返回类型和用于表示失败或者`none`的类型结合起来。例如，下面是`(optionof number)`的等价实现：

```
(define-type MaybeNumber (U Number Boolean))
```

同时，`Boolean`本身也可以是`True`和`False`的联合，在Typed Racket中也确实如此。因此，选择（option）类型更为准确的模拟实现应该是：

```
(define-type MaybeNumber (U Number False))
```

更为一般的，可以定义：

```
(struct: none ())
(define-type (Maybeof T) (U T none))
```

由于由于`none`是新的、独特的类型，不会和其它类型混淆，因此该定义适用于所有类型。它提供给我们与选择类型相同的好处，且我们的值没有被埋入深一层的`some`结构体，而是立即可用。例如`member`，其Typed Racket中的类型是：

```
(All (a) (a (Listof a) -> (U False (Listof a))))
```

如果元素未找到，`member`返回`false`；否则，它将返回从该元素开始的链表（即，链表的第一个元素是期望的元素）。

```
> (member 2 (list 1 2 3))
'(2 3)
```

将其转换为使用`Maybeof`实现，可以写成：

```
(define: (t) (in-list? [e : t] [l : (Listof t)]) : (Maybeof (Listof t))
  (let ([v [member e l]])
    (if v
        v
        (none))))
```

如果元素未找到，它将返回值`(none)`；如果找到了，仍然是返回链表：

```
> (in-list? 2 (list 1 2 3))
'(2 3)
```

这样就无需从`some`容器中取出链表。

15.3.3.3 辨识无标签联合

将值放入联合是一码事；我们还需要考虑如何以类型良好的方式将值从其中取出来。在我们的类ML类型系统中，我们使用程式化的符号——我们的语言中 `type-case` ，ML中的模式匹配——来标识和取出各部分。具体来说，对于代码：

```
(define (safe-to-transport? [a : Animal]) : boolean
  (type-case Animal a
             [armadillo (a?) a?]
             [boa (l) (not (big-one? l))]))
```

在整个表达式中 `a` 的类型保持一致。标识符 `a?` 和 `l` 分别被绑定到布尔类型和数类型的值上， `big-one?` 接收的就是这些类型，而不是 `armadillo` 和 `boa` 。换句话说， `big-one?` 函数的输入类型不可以是 `boa` ，因为根本没有这样的类型。

反之，使用联合类型的话，我们确实有 `boa` 类型。因此，我们遵守对值进行谓词操作将缩小其类型的原则。例如，在 `cond` 的子句

```
[(boa? a) (not (big-one? a))]
```

中，尽管 `a` 的初始类型为 `Animal` ，在通过 `boa?` 测试后，类型检查器会将其类型缩小到 `boa` 的分支，这样 `big-one?` 调用得以通过类型检查。反过来，其在条件表达式剩余部分的类型不是 `boa` ——这里，只剩下 `armadillo` 一种可能。这给类型检查器提出了更高的要求，它需要能测试并识别特定模式（称为条件分割(if-splitting)）；缺了这种能力就无法使用联合类型编程；当然我们可以只识别类ML系统中能识别的模式，也就是模式匹配、 `type-case` 。

15.3.3.4 改造为静态类型

毫不奇怪，Typed Racket使用联合类型。当将现有语言改造为静态类型时，它们尤其有用，因为现有语言（如脚本语言中）的程序没有用类ML类型系统的原则来定义。这种类型改造的通用的原则之一是尽可能多地静态捕获动态异常。当然，检查器最终会让一些程序无法通过检查【注释】，但如果它拒绝太多可以无错运行的程序，开发者不太可能采用它。由于这些程序是在没有考虑类型检查的情况下编写的，因此类型检查器需要以更为激进的方式接受该语言中被认为合理的习惯用法。

> 除非它实现了称为软类型（soft typing）的有趣想法：不拒绝任何程序，而是提供信息告知程序中无法通过类型检查之处。

考虑下面的JavaScript函数：

```
var slice = function (arr, start, stop) {
  var result = [];
  for (var i = 0; i <= stop - start; i++) {
      result[i] = arr[start + i];
  }
  return result;
}
```

它读入一个数组和两个索引，返回这两个索引之间的子数组。例如，`slice([5, 7, 11, 13], 0, 2)` 求得 `[5, 7, 11]` 。

在JavaScript中，开发人员在函数调用时可以自由的省略任意或者所有尾部参数。每个被省略的参数都被赋予特定值 `undefined` ，如何处理这种情形完全由函数决定。例如， `slice` 的典型实现允许用户省略最后一个参数；下面的定义

```
var slice = function (arr, start, stop) {
  if (typeof stop == "undefined")
    stop = arr.length - 1;
  var result = [];
  for (var i = 0; i <= stop - start; i++) {
    result[i] = arr[start + i];
  }
  return result;
}
```

在未给定第三个参数时自动返回到数组结尾的子数组：因此 `slice([5, 7, 11, 13], 2)` 返回 `[11, 13]` 。

在Typed JavaScript【注释】中，程序员可以通过为给定参数指定类型 `U Undefined` 来显式地指明函数可以接受更少的参数，此函数的类型如下：

```
∀ t : (Array[t] * Int * (Int U Undefined) -> Array[t])
```

> 由Arjun Guha等人在布朗（大学）创建。参见我们的网站。

原则上，这意味着表达式 `stop - start` 存在发生类型错误的可能，因为 `stop` 可能不是数。然而，当用户省略该参数时，对 `stop` 的赋值正好将其设为数类型。换句话说，在所有控制路径上，减法发生前 `stop` 都将是数类型，因此该函数能通过类型检查。当然，这要求类型检查器能够对控制流（条件）和状态（赋值）进行推断来确保函数类型正确；而Typed JavaScript可以做到，也因此能允许这样的函数。

15.3.3.4 设计选择

拥有联合类型的语言中，通常有

- 独立的结构体类型（通常用类表示），而不是带有变体的数据类型。
- 用于表示特定类型的特殊（ad hoc）结构体集合。
- 哨兵值（sentinel value）表示失败。

将这种风格的程序转换成满足类ML类型风格的非常费事。因此，许多改造过来的类型系统引入联合类型来减轻类型化过程的负担。

上述三个属性中，第一个相对中立，但是其它两个需要更多讨论。我们以反序依次解决它们。

- 首先处理哨兵值。很多情况下，哨兵应该被替换为异常，但是在很多语言中，抛出异常的代价巨大。因此开发者倾向于区分真正的异常情况——不应该发生——和正常运行中的预期情况。检查元素是否属于链表发现不存在的情况显然属于后者（如果我们已经知道元素是否存在，这个谓词判断就无需进行）。在后一种情况下，使用哨兵是合理的。

  然而，我们需要认识到，在C程序中，未能检测异常的哨兵值是错误——甚至安全缺陷——的常见原因。这点很容易解决。在C中，哨兵值和普通返回值类型相同（或者至少等同于类型相同），而且运行时也没有检查。因此哨兵可以被当作合法的值使用，且不会出现类型错误。这就导致哨兵值 `0` 可以被当作分配数据的地址来使用，从而导致系统崩溃。与之不同，我们的哨兵是真正意义上的新类型，无法用于任何计算。观察到前语言中没有任何函数的输入类型为 `none` ，可以推理出这点。

- 先忽略这里贬义的“特殊”一词，对一组结构体进行不同的分组是否是个好主意？实际上，就算在遵循类ML规范的程序中，当程序员希望刻画一个大宇宙的子宇宙时，也会出现这种分组的情形。例如，ML程序员会使用下面的类型

  ```
  (define-type SExp
    [numSexp (n : number)]
    [strSexp (s : string)]
    [listSexp (l : (listof SExp))])
  ```

  表示s-expression。如果有函数希望操作这些项的某个子集，比如数和数的链表，就必须创建新的类型，然后将值在两种类型之间转换，尽管这两个类型的内部表示完全相同。另一个例子，考虑CPS表达式的集合，这显然是所有可能表达式的一个子集，但如果不得不为其创建新的类型，我们将无法对其使用任何已有的表达式处理程序，比如解释器。

换种说法，联合类型似乎是我们之前见到的ML风格类型系统的合理变种。但是，即使在联合类型中仍有设计选择，它们都有其后果。例如，允许类型系统创建新联合类型吗？允许用户定义（和命名）联合吗？也就是说，允许表达式

```
(if (phase-of-the-moon)
    10
    true)
```

通过类型检查吗（将创建类型 `(U Number Boolean)` ），还是由于其引入了之前未命名并显式标识的类型而将其判定为类型错误？Typed Racket提供的是前者：它将创建真正的临时联合。对于给现有代码引入类型来说，这么做可能更好，因为它更加灵活。但对于写新代码来说，这是否是个好的设计还并不清楚，因为并非程序员期望内的联合会出现，而且无法避免。这给程序语言的设计空间提供了一个未被探索的角落。

15.3.4 名义类型系统与结构类型系统

我们最初的类型检查器中，如果两个类型具有相同的结构，则认为它们是相同的。事实上我们根本没有提供类型的命名机制，因此不清楚有何替代方案。

现在考虑Typed Racket。程序员可以写

```
(define-type NB1 (U Number Boolean))
(define-type NB2 (U Number Boolean))
```

然后写

```
(define: v : NB1 5)
```

假设还定义了函数

```
(define: (f [x : NB2]) : NB2 x)
```

然后用 `v` 调用 `f` ，即 `(f v)` ：该调用应该通过类型检查吗？

有两种完全合理的解释。一种是说 `v` 被声明为类型 `NB1` ，与 `NB2` 名称不同，因此应该被当作不同类型，所以该调用应导致错误。这种系统被称为名义的（nominal），因为类型的名字对于确定类型是否相等极为重要。

与之对应，另一种解释是说因为 `NB1` 和 `NB2` 结构相同，因此开发者无法写出在这两种类型的值上表现的不同的程序来，所以它们应该被视为相同。【注释】这种类型系统被称为结构的（structural），将允许上面的程序通过检查。（Typed Racket遵循结构类型的规范，理由同样是减少导入现有动态类型代码的负担，这些Racket代码通常是以结构解释为模型编写的。事实上，Typed Racket中 `(f v)` 不仅能通过类型检查，而且打印出的返回类型为 `NB1` ，无视 `f` 返回值的类型注解！）

> 如果特别小心，你会注意到被认为相同和实际相同之间是有区别的。这里不会涉及该问题，但请考虑编译器作者选择值的表示时其影响是啥，尤其在允许运行时获取值的静态类型的语言中。

名义和结构类型之间的区别在面向对象语言中是最常见的争议，后面将简要回顾这个问题。然而，这里的重点是要说明这些问题本质上并不关乎“对象”。任何允许命名类型的语言——出于程序员精神健康的需要，也就是所有的语言了——都要应付此问题：命名只是方便起见，还是说所选的名字是被认为是有意义的？选择前者导致结构类型，选择后者导致名义类型。

15.3.5 交叉类型

我们刚探索了联合类型，很自然的就会想到有没有交叉（intersection）类型呢。确实有。

如果联合类型指（该类型的）值属于这个联合中某个类型，交叉类型显然意味着该值属于交叉中的所有类型：合取，或“且”。这可能看起来很奇怪：值怎么可能属于多种类型呢？

用具体例子回答，考虑重载函数。例如，某些语言中 `+` 即可操作数，也能操作字符串；传入两个数它返回数，传入两个字符串它返回字符串。这种语言中，`+` 的类型应该是什么呢？不是 `(number number -> number)`，因为那样它将不能用于字符串；同样的原因，也不是 `(string string -> string)`。甚至它也不是

```
(U (number number -> number)
   (string string -> string))
```

因为 `+` 不仅仅是这些函数之一：实际上它（同时）是这两者。我们可以认为其类型是

```
((number U string) (number U string) -> (number U string))
```

这说明它的每个参数和返回值都只能是这两种类型之一，而不同时为两者。但是，这样做会导致精度损失。

思考题

> 这种类型以何种方式损失精度？

观察到，对于这个类型，所有函数调用的返回值类型均为 `(number U string)`。因此，对于每个返回值都必须区分数和字符串，不然我们将得到类型错误。所以，尽管我们知道给定两个数参数将返回数结果，但这种信息在类型系统中丢失了。

更巧妙的是，这个类型允许独立的选择每个参数的类型。因此，根据该类型，`(+ 3 "x")` 也是合法的（且其返回值类型为 `(number U string)`）。但我们描述的加法操作当然没有对这组参数定义过！

因此描述这种加法的更为合适的类型是

```
(^ (number number -> number)
   (string string -> string))
```

这里的 `∧` 让人联想到逻辑上的合取操作符。这允许函数用两个数或者两个字符串进行调用，其它的则不允许。使用两个数调用返回数类型；使用两个字符串调用返回字符串类型；除此之外没有其它合法调用了。这刚好对应于我们期望的重载行为（有时也称为特设多态(ad hoc polymorphism)）。请注意这只能处理有限数量重载的情况。

15.3.6 递归类型

学过联合类型之后，值得讨论一下我们原来遇到过的递归数据类型表达式。如果接受变体作为类型构造器，我们可以将递归类型写作它们的联合吗？例如就 `BTnum` 来说，能否将它描述成等价于

```
((BTmt) U (BTnd number BTnum BTnum))
```

的类型吗，其中`BTmt`是零参数的构造器，而`BTnd`是三参数的？不过，这三个参数的类型是什么？按上面所写的类型，`BTnum`要么是类型语言内建的（这不能令人满意），要么是未绑定的。也许我们要的是

```
BTnum = ((BTmt) U (BTnd number BTnum BTnum))
```

问题是这个方程没有明显解法（还记得`ω`吗？）。

这种情况我们讨论值的递归时就熟悉过。那时，我们发明了递归函数构造器（并展示了其实现）来规避这个问题。这里我们同样需要递归类型构造器。按惯例它被称为`μ`（希腊字母“缪”）。有了它，我们可以将上面的类型写做

```
μ BTnum : ((BTmt) U (BTnd number BTnum BTnum))
```

`μ`是绑定构造；它将`BTnum`绑定到后面写的整个类型上，包括对`BTnum`自身的递归绑定。实践中，整个递归类型就是我们希望得到的称为`BTnum`的类型：

```
BTnum = μ BTnum : ((BTmt) U (BTnd number BTnum BTnum))
```

尽管这看起来像是循环定义，但请注意，右侧的`BTnum`不依赖于等式左侧的那个：即，我们可以将其重写为

```
BTnum = μ T : ((BTmt) U (BTnd number T T))
```

换句话说，`BTnum`的这个定义可以被认为是语法糖，可以在程序的各个地方替换使用，无需担心无限回归的问题。

语义层面上，对`μ`绑定的类型的意义有两种截然不同的思考方式：它们可以被解释为同构递归（isorecursive）或等价递归（equirecursive）。然而其中区别很微妙，超出了本章范围。【注释】只需理解递归类型可以被视为等同于它的展开。例如，我们定义数的链表类型为

```
NumL = μ T : ((MtL) U (ConsL number T))
```

于是有

```
  μ T : ((MtL) U (ConsL number T))
= (MtL) U (ConsL number (μ T : ((MtL) U (ConsL number T))))
= (MtL) U (ConsL number (MtL))
        U (ConsL number (ConsL number (μ T : ((MtL) U (ConsL number T)))))
```

以此类推（同构和等价递归之间的区别正是在相等性的概念上：是定义上的相等性还是同构意义上的）。每一步中，我们将参数`T`替换成整个类型。和值的递归一样，它的意思是需要时我们可以“获得另一个”`ConsL`构造。换种说法，链表的类型可以写成零或任意多元素的联

合；这等价于包含零个、一个或任意个元素的类型；以此类推。任何数的链表都（恰好）符合这些类型。

> Pierce的书中对此解释的非常好。

注意到，即使基于对于`μ`的这种非正式理解，我们已经可以给`ω`进而`Ω`提供类型。

练习题

> 描述`ω`和`Ω`的类型。

15.3.7 子类型

想象我们有一个典型的二叉树定义；简单起见，我们假设值为数。使用Typed Racket写该定义：

```
#lang typed/racket

(define-struct: mt ())
(define-struct: nd ([v : Number] [l : BT] [r : BT]))
(define-type BT (U mt nd))
```

考虑二叉树树具体的值：

```
> (mt)
- : mt
#<mt>
> (nd 5 (mt) (mt))
- : nd
#<nd>
```

注意每个构造器构造出其自己对应类型的值，而不是类型`BT`的值。但是考虑`(nd 5 (mt) (mt))`：`nd`的定义声明其子树必须为`BT`类型，但是我们可以给它传递`mt`类型的值。

显然，使用`mt`和`nd`来定义`BT`并不是巧合。但是，它确实表明在进行类型检查时，不能只检查构造函数的相等性，至少对我们目前有的东西不行。相反，我们必须检查一种类型“适用于”另一种。这种行为也被称为子类型化（subtyping）。

子类型化的本质定义一个关系，通常用`<:`表示，它关联一对类型。当给定类型为`S`的值时，那么它同时也是类型`T`时我们就可以说`S <: T`：换句话说，子类型化将可替代性的概念形式化的表达出来（即，任何期望类型`T`的值的地方，都可以被替换成类型为`S`的值）。当该关系成立，`S`被称作子类型，`T`被称作超类型。使用子集去解释这点是很有用的（通常也是准确的）：如果`S`的值是`T`的一个子集，那么期望接受类型为`T`的值的表达式不应该拒绝只接受`S`中的值。

子类型化对类型系统有着深远影响。我们需要审视每种类型，并理解它和子类型化之间的相互作用。对于基本类型，这通常比较明显：数字、字符串等的这种不想交类型，彼此无关（存在一些语言，使用一种基本类型表示其它基本类型——例如，某些脚本语言中，数只不过是特殊写法的字符串，还有些语言中，布尔值不过就是数——这些语言中，基本类型之间会存在关系，但是这种情况并不常见。）。但是，我们必须考虑子类型化和每个复杂类型构造器之间的关系。

事实上，甚至我们关于类型的表述也需要改变。假设我们有一个类型为 `T` 的表达式。通常我们会说它产生类型为 `T` 的值。现在，我们需要小心的说它产出的值最多是类型 `T` ，因为它可能只产出类型 `T` 的一个子类型的值。因此每个对类型的引用都隐含的笼罩在对可能的子类型的引用中。为避免纠缠我会控制不这样做，但要小心，不把这种隐含的解释放在心上可能招致推理错误。

15.3.7.1 联合

让我们来看看联合和子类型化会发生什么作用。显然，每个子联合是整个联合的子类型。在我们之前的示例中，每个 `mt` 类型的值显然也都是 `BT` 的；这同样适用于 `nd` 。因而，

```
mt <: BT
nd <: BT
```

于是， `(mt)` 的类型也为 `BT` ，因此表达式 `(nd 5 (mt) (mt))` 类型良好，且有类型 `nd` ——因此，也是类型 `BT` 。一般来说，

```
S <: (S ∪ T)
T <: (S ∪ T)
```

（我们写了两个看上去差不多的的规则，这是为了明确说明子类型处在联合中的哪“一边”并不重要）。它的意思是 `S` 的值可以被认为是 `S ∪ T` 的值，因为任何类型为 `S ∪ T` 的表达式都确实包含类型 `S` 的值。

15.3.7.2 交叉

既然到了这一步，我们也简要的看一下交叉类型。正如你可能想象的那样，交叉类型表现出双面行为：

```
(S ∧ T) <: S
(S ∧ T) <: T
```

方便起见，使用集合的解释：如果值即属于 `S` 也属于 `T` ，显然，它可以是它们中的任意类型。

思考题

> 为什么下面两者无效？
>
> 1. `(S ∪ T) <: S`
> 2. `T <: (S ∧ T)`

第一条无效是因为类型 `T` 的值是 `(S ∪ T)` 中完全合法的值。例如，数是类型 `(string U number)` 的一员。然而，数不可以在需要类型为 `string` 的时候被使用。

至于第二条，类型 `T` 的值一般来说不是类型 `S` 的值。任何希望类型 `(S ∧ T)` 消费者希望其能够既作为 `T` 也作为 `S` ，因此这这第二条是不对的。例如，给定前面见过的重载的 `+` ，如果 `T` 为 `(number number -> number)` ，那么该类型的函数无法对字符串进行处理。

15.3.7.3 函数

再看一个类型构造器：函数。我们需要决定子类型化时，其中一个类型可以为函数的情况。通常我们认为函数和其它类型不相交，因此我们只需要考虑函数类型作为另一个函数类型的子类型的情况：也既，何时会满足下面式子？

```
(S1 -> T1) <: (S2 -> T2)
```

> 我们也见过参数化数据类型。这个版本中，对它们的子类型化的探索作为留给读者的练习。

方便起见，我们称类型 `(S1 -> T1)` 为 `f1` ，`(S2 -> T2)` 为 `f2` 。问题就变成了，如果一个表达式类型为 `f2` ，何种情况下给安全的给其传递 `f1` 类型的函数？使用子集解释来考虑这个问题比较容易。

考虑 `f2` 类型的使用。它返回值的类型为 `T2` 。因此函数调用所在的上下文中所需的值类型应该满足类型 `T2` 。显然，如果 `T1` 和 `T2` 相同，那么在该上下文中使用 `f2` 也能通过类型检查；类似的，如果 `T1` 包含 `T2` 值的一个子集，也是可以的。唯一的问题是，如果 `T1` 的值比 `T2` 多，该上下文将可能遭遇非期望的值，从而导致未定义行为。换句话说，我们需要 `T1 <: T2` 。注意其中符号的“方向”与函数类型中符号的方向相同；这被称为协变（covariance，两者在相同的方向上变化）。这也许正是你期望的。

出于同样的原因，你可能认为参数位置也出现协变：即 `S1 <: S2` 。能猜到你会这样想，但它是错的。让我们看看为什么。

对 `f2` 类型的函数进行调用需要提供类型为 `S2` 的值的参数。假设我们将函数替换为类型 `f1` 的。如果 `S1 <: S2` ，这意味着新的函数仅接受类型 `S1` 的一个子集中的值——严格子集。这意味着对于某些值，替换的函数的行为是未定义的，这导致未定义的行为。为避免此，需要假定相反的方向：即替代函数应该至少能接收原函数能够接收的那些值。因此我们需要 `S2 <: S1` ，我们说该位置是逆变（contravariant）的：它和子类型化方向相反。

综合这两个发现，我们得到函数子类型化的规则（对于方法也一样）：

```
(S2 <: S1) and (T1 <: T2) => (S1 -> T1) <: (S2 -> T2)
```

15.3.7.4 实现子类型

当然，这些规则假定我们已经修改了类型检查器遵循子类型化的要求。子类型化的本质是说，如果有一个表达式 `e` ，其类型为 `S` ，且 `S <: T` ，那么 `e` 具有类型 `T` 。虽然这听上去很直观，但它是有问题的，出于两个原因：

- 到目前为止，我们所有的类型规则都是语法驱动的，这使我们可以编写一个递归下降的类型检查器。但是，现在我们有一个适用所有表达式的规则，我们不知道怎样去应用这条规则了。
- 可能存在很多级别的子类型。这使得何时“停止”子类型化不再是个显而易见的问题。特别是，现在表达式可以有很多可能的类型，在类型检查能计算出表达式的类型之前；如果我们返回了“错误”的，可能会遇到类型错误（因为它不是上下文期望的类型），尽管这时候可能存在一些其它的能够满足上下文需求的类型。

这两个问题指出的是，我们这里给出的关于子类型化的描述根本上来说是声明性的：我们在说它应该是怎样的，但是没有将这种说明转换成算法。对于每个实际的静态类型语言，将其转换成算法的子类型化时总会遇到多或少被认为是有趣的问题：一种实现类型检查器的实际算法（理想情况下，该类型检查器能让所有声明机制下被认为是有效的程序通过类型检测，也即，既正确又完全）。

15.3.8 对象类型

早先我们提到过，对象类型通常分为两个阵营：名义的（nominal）和结构的（structural）。名义类型对于大多数程序员来说在Java的学习中应该已经熟悉，所以这里不再多说。对象的结构类型是说对象的类型本身就是一个结构化的对象，包含字段的名字及它们的类型。例如，有两个方法—— `add1` 和 `sub1` ——的对象，其类型将是：

```
{add1 : (number -> number), sub1 : (number -> number)}
```

（为方便引用，我们称这个类型为 `addsub` 。）类型检查的将沿着可预见的路径行进：对于字段的访问，我们只需确保字段存在，并将该字段的声明类型用于它解引用得到的表达式上；对于方法调用，我们不仅需要确保对应成员存在，还要确保其类型正确。到目前为止，十分简单。

对象类型会因为很多原因变得十分复杂：

> 尽管有点过时，但是Abadi和Carelli的*A Theory of Objects*仍然很重要，因为这整本书都专注于该话题。Bruce的*Foundationos of Object-Oriented Languages: Types and Semantics*更为现代，也提供了更温和的阐述。Perce则漂亮的覆盖了所有必要的理论。

- 自引用。 `self` 的类型是什么？它必须拥有和自己那个对象相同的类型，由于任何可以

从“外部”施加到对象上的操作也可以通过`self`在“内部”施加给它。这意味着对象是递归类型。

- 访问控制：私有、公共和其它限制。这导致对象类型“外部”和“内部”之间的区别。
- 继承：不仅需要为父对象指定类型，还需要考虑继承路径上哪些东西可见，这和“外部”可见的东西又有区别。
- 多重继承和子类型之间的相互作用。
- 像Java这样的语言中类和接口之间的关系存在运行时成本。
- 赋值。
- 类型转换。
- Snakes on a plane。

等等。其中的一些问题会因为名义类型而简化，因为给定类型名我们就可以确定有关其行为的所有信息（类型声明实际变成了一个字典，从中我们可以查询关于对象的描述），这也是赞成名义类型的一个论据。

> 注意Java的方法不是构建名义类型系统的唯一方法。我们已经讨论过，Java的类系统不必要的限制了程序员的表达能力；相对的，Java的名义类型不必要的将类型（接口描述）和实现混为一谈。因此，是可以有比Java中的好得多的名义类型系统的。例如，Scala在这个方向上迈出了一些很重要的步子。

对这些问题进行充分论述需要在更大的设计空间中讨论。现在我们限制自己到一个有趣的问题。记得我们说过子类型化迫使我们考虑每个类型构造器吗？对象的结构类型要再引入一类：对象类型构造器。因此我们必须了解它与子类型化之间的相互作用。

在那之前，让我们确保我们理解对象类型到底意味着什么。考虑上面的`addsub`类型，其中列举了两个方法。何种对象可以给定此类型？显然，恰好拥有这两个方法且类型符合的对象才符合条件。同样明显的是，如果一个对象只包含这两个方法中的一个，不管它还包含有其它什么，都不符合条件。其中短语“不管它还包含其它什么”是要强调的。如果对象表示的是算术包，除了这两个方法之外，它还包含`+`和`*`呢？这种情况下，我们当然得到了一个能供应上面两个方法的对象，因此该算术包确实有类型`addsub`。不过将其作为类型`addsub`使用时，其它方法应该不可用。

下面我们写下这个包的完整类型，称之为`as+*`：

```
{add1  : (number -> number),
 sub1  : (number -> number),
 +     : (number number -> number),
 *     : (number number -> number)}
```

前面讨论过了类型`as+*`的对象应该也允许被声明称类型`addsub`，这意味着它可以替换到在上下文中期望是`addsub`类型的值。换句话说，我们刚才的意思其实是`as+* <: addsub`：

```
{add1  : (number -> number),           {add1 : (number -> number),
 sub1  : (number -> number),        <:  sub1 : (number -> number)}
 +     : (number number -> number),
 *     : (number number -> number)}
```

这可能乍一看令人困惑：我们说过子类型化遵从集合包含关系，因此我们期望小的集合在左侧而大的集合在右侧。可这里，好像“大的类型”（至少在字符数量的意义上是）在左侧而“小的类型”在右侧。

为了理解为什么这是正确的，发展出类型“越大”，其包含的值越少这种直觉会很有帮助。左侧的每个对象都含有四个方法，而且其中包含了右侧的那两个方法。但是，有很多对象有右侧的两个方法，但是不包含左侧那另外的两个方法。如果我将类型看作对可接受值形状的约束的话，“更大”的类型表明它给定了更多约束，因此会导致更少的值。这样，尽管类型可能看上去大小关系不对，但是它们所包含的值的集合的大小关系是正确的。

更一般地，从对象中删除字段，我们将获得超类型。这被称为宽度子类型化（width subtyping），因为子类型“更宽”，我们通过调整对象“宽度”来移动到更上层的类型。我们甚至可以在名义类型的Java世界中看到这点：当沿着继承链上溯时，类中的方法和字段越来越少，直到 `Object` ，它是所有类的超类型，其中包含最少的字段和方法。因此对于Java中的任意类类型 `C` ，满足 `C <: Object` 。

正如你期望的那样，还有一种重要的子类型形式，在给定成员内部。就是说任何特定的成员都可以归入相应位置的超类型。处于显而易见的原因，这种形式的子类型化被称为深度子类型化（depth subtyping）。

> 有时，缩小（narrowing）和拓宽（widening）的使用方式会让人疑惑，它看上去好像用反了一样。拓宽是指从子类型转到超类型，因为它是从一个“较窄”（较小）的集合到一个“较宽”（较大）的集合。这些术语是独立演化而来的，很不幸，并不一致。

练习题

> 构造两个深度子类型化的例子。其中一个，给定字段为对象类型，使用宽度子类型化去取该字段的子类型。另一个例子中，给定字段为函数类型。

Java中限制了深度子类型化，它倾向于类型在对象层次结构中保持不变，因为这对传统的赋值操作来说是安全的。

宽度和深度子类型化的结合包含了对象子类型化中大部分最有趣的情形。然而，仅实现这两个子类型化的类型系统会不必要的招致程序员的恼火。其它方便的（而且数学上必须的）规则还包括可以排列名称的能力、反身性（每个类型是其自己的子类型，因为将子类型关系解释为 `⊆` 很方便）和传递性。类似Typed JavaScript的语言使用了所有这些特性为程序员提供最大的灵活性。

16 动态地检查程序中的不变量：契约

类型系统提供丰富且有价值的方式来表示程序不变量。（这两句请校对修改）然而，它们也代表了一个重要的权衡，因为并非所有程序的非平凡属性都可以被静态验证。【注释】此外，即使某个属性可以设计静态方法解决，注释和计算复杂度的负担也可能过大。因此，我们所关心的一些属性不可避免地只能被忽略或在运行时解决。本章我们来讨论运行时检查不变量。

> 这是一个形式上的属性，被称为赖斯定理。

实际上，每种编程语言都包含某种形式的断言机制，使程序员能够编写比语言的静态类型系统所允许的更丰富的属性。在没有静态类型的语言中，这些属性可能以简单的类型断言开始：例如，某个参数是否为数。然而，断言语言通常是整个编程语言，因此任何谓词都可以用作断言：例如，某个加密包的实现可能希望确保某些参数通过素数测试，或者某个平衡二叉搜索树可能想要确保其子树确实是平衡且有序的。

16.1 以契约实现谓词

因此很容易想到如何实现简单的契约（contract）。【注释】契约实现了谓词。它读入一个值并将谓词应用于该值。如果值能通过谓词，则契约原样返回该值；如果不能，则谓词会报告错误。其行为只能是返回原值或报错：它不应以任何方式更改值。简而言之，对于能通过谓词的值，契约本身就是恒等函数。

> 下面我们将使用 `#lang plai` 语言，原因有两个。首先，这更好地模拟了动态类型语言编程。其次，为了简单起见，我们会将类型声明写成契约，但是在静态类型语言中，这将由类型检查器处理，而我们无法看到运行时行为。从效果来看，“关闭”类型检查器会更容易。然而，即使在静态类型的世界里，契约也是非常有意义的，因为它们增强了程序员可以表达的不变量。

这些可以编码成如下的函数：

```
(define (make-contract pred?)
  (lambda (val)
    (if (pred? val) val (blame "violation"))))

(define (blame s) (error 'contract "~a" s))
```

契约的例子：

```
(define non-neg?-contract
  (make-contract
   (lambda (n) (and (number? n)
                    (>= n 0)))))
```

（在静态类型语言中，检查`number?`当然是不必要的，因为它可以由使用契约的函数的类型编码并静态检查！）假设我们计算平方根时要确保不会得到虚数；可以这么写

```
(define (real-sqrt-1 x)
  (sqrt (non-neg?-contract x)))
```

在很多语言中，断言是写作语句而不是表达式，所以另一种编写方式是：

```
(define (real-sqrt-2 x)
  (begin
    (non-neg?-contract x)
    (sqrt x)))
```

（在某些情况下，这种形式更清晰，因为它在函数的开始部分清晰地声明了参数的期望值。它还确保参数只被检查一次。实际上，在某些语言中，契约可以写入函数头部中 ，从而改善函数界面中能给出的信息。）现在，如果将`real-sqrt-1`或`real-sqrt-2`应用于`4`，则它们产生`2`，但如果应用于`-1`，则会引发违反契约的错误。

16.2 标签、类型和对值的观察

到这里我们已经重现了大多数语言中断言系统的本质。还有什么要讨论的？我们假设手上的语言不是静态类型的。那么我们想要编写的断言至少要能重现传统的类型不变量，如果不是更多的话。前述的`make-contract`可以覆盖所有标准类型的属性，比如检查数、字符串等等，假设语言提供了合适的谓词，或者可以从已有的谓词中构造出来。是这样吗？

回想一下，即使我们最简单的类型语言也不仅仅是类似数的基本类型，还包含构造类型。尽管其中的一些，如链表和向量，似乎并不是很难，但一旦涉及赋值、性能和责备（这一点将在后面讨论）挑战就来了。然而，函数就很难处理了。

作为示例，我们来看这个函数：

```
(define d/dx
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x 0.001))
            (f x))
         0.001))))
```

其静态类型是：

```
((number -> number) -> (number -> number))
```

（它读入一个函数，并生成由其派生的另一个函数。）假设我们想用契约来处理这种情况。

根本的问题是，在大多数语言中，我们无法直接将其表示为谓词。大多数语言的运行时系统关于值的类型存储了非常有限的信息——对于迄今为止我们所看到的类型，这些有限的信息可以用不同的名称来描述；传统上它们被称为标签（tag）。【注释】有些情况下，标签与我们认为的类型相符：例如，数会带上标签，将其标识为数（甚至可能是某种特定类型的数）、字符串带的标签将其标识为字符串，等等。因此，我们可以基于这些标签的值来编写谓词。

> 已经有一些工作试图保存丰富的类型信息，从源程序到较低的抽象层次、一直到汇编语言，但这些都是研究工作。

当我们处理结构化值时，情况就复杂了。向量将会带有标签声明它是向量，但不会指明它的元素是什么类型的值（而且它们甚至可能都不是同一类型）；不过，程序通常也可以获得向量的大小，从而遍历向量来收集此信息。（然而，关于结构化值后面有更多讨论。）

思考题

> 编写契约，检查只包含偶数的链表。

这就是：

```
(define list-of-even?-contract
  (make-contract
   (lambda (l)
     (and (list? l) (andmap number? l) (andmap even? l)))))
```

（同样，请注意，如果我们静态地知道这是数的链表，则无需问前两个问题。）类似地，对象可能只是将自己标识为对象，而不提供其他信息。但是，在允许对对象结构进行反射（reflection）的语言中，契约仍可以收集它所需的信息。

然而，在任何语言中，当遇到函数时就出问题了。我们一般将函数的类型理解为包含其输入和输出的类型，但是对运行时系统，函数只是带有函数标签的不透明对象，可能还有一些非常有限的元数据（如函数的参数数量）。运行时系统甚至难以分辨函数是否读入和生成函数——而非其他类型的值——更不用说它是否读入并生成 `(number -> number)` 类型的函数。

这个问题很好地体现在JavaScript的（错误命名的） `typeof` 运算符中。传给其数或字符串等基本类型的值， `typeof` 会返回其名字所示的字符串（例如 `"number"` ）。对于对象，它返回 `"object"` 。最要命的是，对于函数它返回 `"function"` ，没有额外的信息。

> 出于这个原因， `typeof` 对这个操作符来说可能是个糟糕的名字。它应该被称为 `tagof` ，为未来的JavaScript静态类型系统提供了真正的 `typeof` 留出空间。

总而言之，这意味着当遇到函数时，函数契约只能检查它是否的确是函数（如果不是，那显然是错误的）。它无法检查有关该函数的定义域和值域【讨论：翻译成输入输出(后同)会不会更合适？】的任何信息。我们要放弃吗？

16.3 高阶契约

为了确定要做什么，我们先回忆一下契约最初提供了什么保证。在前述的`real-sqrt-1`中，我们要求参数是非负的。然而，只有在实际使用`real-sqrt-1`的情况下才会进行检查，并且仅检查实际传入的值。例如，如果程序包含片段

```
(lambda () (real-sqrt-1 -1))
```

但该thunk一直没被调用，那么程序员永远不会看到这里的契约违反。事实上，可能在程序的这次运行中没有调用此thunk，但在后一次运行中调用到了；因此，该程序包含一个潜在的契约错误。出于此原因，通常最好用静态类型来表示不变量；但在使用契约时，我们明白，仅当程序执行到相关位置时，我们才会收到错误通知。

这是有用的见解，因为它为我们的函数问题提供了解决方案。对于指明的函数值，我们立即检查它真的是函数。但是，我们不会忽略定义域和值域的契约，而是延迟进行。我们在函数（每次）实际作用于某个值时检查定义域契约，并在函数实际返回值时检查值域契约。

这显然和`make-contract`不是一种模式。因此，我们给`make-contract`起个更具描述性的名称：它检查即时的（immediate）契约（即当前可以完整检查的契约）。

> 在Racket契约系统中，即时契约被称为扁平的（flat）。这个术语有点误导，因为它们也可以保护数据结构。

```
(define (immediate pred?)
  (lambda (val)
    (if (pred? val) val (blame val))))
```

相比之下，函数契约读入两个契约作为参数——分别表示对定义域和值域的检查——并返回谓词。这个谓词作用于需要满足契约的值。首先，它会检查给定的值实际上是函数：这部分仍然是即时的。然后，我们创建一个代理（surrogate）函数，由它来应用“剩余的”契约——检查定义域和值域——但其他方面与原函数行为相同。

创建代理这一行为背离了传统的断言机制，也就是只是简单地检查值、保持值的独立。相反，对于函数，如果想要检查契约，我们必须使用新创建的代理。因此，一般来说我们需要创建封装函数，它会读入契约和值，并创建该值的保护版本：

```
(define (guard ctc val) (ctc val))
```

一个非常简单的例子，假设我们要用数契约包装`add1`函数（通过稍后定义的`function`，函数契约的构造函数）：

```
(define a1 (guard (function (immediate number?)
                            (immediate number?))
                  add1))
```

我们希望`a1`本质上绑定到以下代码：

```
(define a1
  (lambda (x)
    (num?-con (add1 (num?-con x)))))
```

其中`(lambda (x) ...)`是代理；它会`add1`的调用之处前后调用数值契约。回忆一下，在没有违规的情况下，契约的行为就是恒等函数，所以这个程序在不违规的情况下行为于`add1`完全相同。

为了达到此目的，我们使用下面的`function`定义。【注释】请记住，我们还必须确保给定的值真的是函数（这里的`add1`的确是，这一点可以立即检查，这也是为什么在我们将代理绑定到a1时此项检查已经消失的原因）：

```
(define (function dom rng)
  (lambda (val)
    (if (procedure? val)
        (lambda (x) (rng (val (dom x))))
        (blame val))))
```

> 简单起见，我们这里假设单参数函数，不过扩展到多参数的情况很简单。事实上，更复杂的契约甚至可以检查参数之间的关系。

要理解这是如何工作的，我们来替换参数。为了保持代码可读性，我们先构造`number?`契约检查器，并将其命名：

```
  (define num?-con (immediate number?))
= (define num?-con
    (lambda (val)
      (if (number? val) val (blame val))))
```

回到`a1`的定义。我们先调用`guard`：

```
(define a1
  ((function num?-con num?-con)
   add1))
```

接下来调用函数契约的构造函数：

```
(define a1
  ((lambda (val)
     (if (procedure? val)
         (lambda (x) (num?-con (val (num?-con x))))
         (blame val)))
   add1))
```

调用左括号-左括号-lambda得：

```
(define a1
  (if (procedure? add1)
      (lambda (x) (num?-con (add1 (num?-con x))))
      (blame add1)))
```

请注意，这一步会检查被保护的值的确是函数。因此我们得到

```
(define a1
  (lambda (x)
    (num?-con (add1 (num?-con x)))))
```

这正是我们想要获得的代理，对于不违规的调用，其行为就是 `add1` 。

思考题

> 有多少种方式可以违背上述的`add1`契约？

三种方式，分别对应于三个契约构造函数：

1. 被封装的值可能不是函数；
2. 被封装的是函数，它可能被作用于不为数的值；或者
3. 被封装的是函数，输入也是数，但其返回值不是数类型。

练习

> 编写示例实现这三种违规行为，并观察契约系统的行为。你能改进错误信息以更好地区分这些情况吗？

同样的封装技术也适用于`d/dx`：

```
(define d/dx
  (guard (function (function (immediate number?) (immediate number?))
                   (function (immediate number?) (immediate number?)))
         (lambda (f)
           (lambda (x)
             (/ (- (f (+ x 0.001))
                   (f x))
                0.001)))))
```

练习

> 违反此契约的方式有七种，分别对应于七个契约构造函数。根据需要，传入（错误的）参数或修改代码，以违反它们中的每一个。是否可以改进错误报告，以正确识别每种违规行为？

请注意，嵌套函数的契约推迟了两处即时契约的检查，而不是一处。这符合我们的期望，因为即时契约只能报告实际值的问题，所以直到应用于实际值之前，它们无法报告任何错误。但是，这确实意味着“违规”这个概念很微妙：传递给`d/dx`的函数值可能的确违反了契约，但这类违规只有在传递或返回数值之后才会被观察到。

16.4 便捷语法

之前我们看到了两种扁平契约的使用风格，分别由`real-sqrt-1`和`real-sqrt-2`体现。这两种风格各有缺点。后者让人联想到传统的断言系统，它不能用于高阶值（函数），因为被封装的值才需要检查。（当然，传统的断言系统只处理扁平契约，所以它们忽略了这个细微的差别。）前者将值的使用放与契约之中，理论上这可行，但有三个缺点：

1. 开发人员可能会忘记封装某些使用。
2. 契约在每次使用中都会被检查一次，在多次使用时这是浪费。
3. 程序混合了契约检查和其功能行为，降低了可读性。

幸运的是，一般情况下，明智地使用语法糖就可以解决此问题。例如，假设我们要将契约附加到函数的参数上，那么开发人员可以这么编写：

```
(define/contract (real-sqrt (x :: (immediate positive?)))
  (sqrt x))
```

意图是用`positive?`来保护`x`，但只在函数调用时只执行一次检查。这应该转化为：

```
(define (real-sqrt new-x)
  (let ([x (guard (immediate positive?) new-x)])
    (sqrt x)))
```

也就是说，宏为每个标识符生成新名称，然后将用户给出的名称关联到新名称的封装版本。这个宏的实现如下：

```
(define-syntax (define/contract stx)
  (syntax-case stx (::)
    [(_ (f (id :: c) ...) b)
     (with-syntax ([(new-id ...) (generate-temporaries #'(id ...))])
       #'(define f
           (lambda (new-id ...)
             (let ([id (guard c new-id)]
                   ...)
               b))))]))
```

有了这些（语法上的）便利，契约语言的设计师可以提高契约使用的可读性、效率和健壮性。

16.5 扩展到复合数据结构

正如我们已经讨论过的，将契约扩展到结构化数据类型（如链表、向量和用户定义的递归数据类型）似乎很容易。唯一需要的是提供适当的运行时观察值。通常事情就是这样，语言提供某种精度的类型。例如，正如我们之前讨论过的，支持数据类型的语言不需要类型谓词，但仍然会提供谓词来区分变体；这种情况下，类型级别的“契约”检查最好（也许必须）留给静态类型系统，而由契约来断言更精确的结构特性。

但是，这种策略可能会遇到严重的性能问题。例如，假设我们编写了平衡二叉搜索树，能以对数渐近时间（相比树的大小）实现插入和查找。接下来我们将树封装在合适的契约中。遗憾的是，仅检查契约就会访问整个树，从而用去线性时间！因此，理想情况下更好的策略是，构建树的时候就（以增量方式）完成契约检查，查找时则不需要再次检查。

更糟的是，平衡和搜索树顺序都是递归属性。因此原则上，它们附加于每个子树上，所以每次递归调用都需要作用。在插入过程中，由于插入是递归的，将在每个访问的子树上检查契约。在大小为 t 的树中，契约谓词应用于 $\frac{t}{2}$ 元素的子树，然后应用于 $\frac{t}{4}$ 元素的子子树，依此类推，在最坏情况下，会访问总数为 $\frac{t}{2}+\frac{t}{4}+...+\frac{t}{t}$ 的元素——使我们预期的对数时间插入过程花费线性时间。

对这两个例子，许多情况下都可以采用措施缓解。每个值都需要与它已经通过的一组契约相关联（或内部存储，或存储于散列表中）。然后，当需要调用契约时，首先检查它是否已被检查过，如果有，则不再检查。这实质上是将契约检查记忆化（memoization），从而减少检查的算法复杂性。当然，对记忆化而言，最好值是不可变的。如果这些值可能发生变化，并且契约执行任意计算，那么此优化可能无法做到可靠。

检查数据结构还有一个微妙的问题。作为例子，考虑我们之前编写的检查数链表中所有值均是偶数的契约。假设我们已经用契约封装了链表，但只对链表的第一个元素感兴趣。当然，我们检查了列表中的所有值，这可能需要很长时间。但更重要的是，用户可能会争辩说，报告链表第二个元素违规的行为本身违反了我们对契约检查的期望，因为我们并未实际使用该元素。

这意味着推迟检查某些值，即使它们可以即时被检查。例如，可以将整个链表转换为包含延时检查的封装值，每个值仅在访问时被检查。这种策略可能很有吸引力，但将其编码并不简单，尤其当存在别名的情况下会遇到问题：如果两个不同的标识符引用同一链表，一个有契约保护而另一个没有，我们必须确保它们都按预期运行（这通常意味着我们不能在链表中存储任何可变状态）。

16.6 契约和观察

契约实现还有一个奇怪的普遍问题，而复杂数据加剧了这个问题。之前，我们抱怨说检查函数的契约很难，因为我们没有足够的能力去观察：我们可以检查的只是值是否是函数。在真实的语言中，数据结构的问题其实是相反的：我们有太多的观察能力。例如，如果我们实施延迟检查链表的策略，则很可能需要使用某个结构体来保存实际列表，并修改 `first` 和 `rest` ，以此（检查契约后）获取结构体中的值。但是，像 `list?` 这样的函数现在可能返回 `false` 而不是 `true` ，因为结构体不是链表；因此， `list?` 需要绑定到新函数上，遇到这些特殊的表示链表的延迟契约结构体也返回 `true` 。但契约系统作者还需要记得解决 `cons?` 、 `pair?` ，天知道还有多少其他函数都可以进行观察。

一般来说，有一个观察基本上不可能“修复”：`eq?` 。通常情况下，每个值 `eq?` 它自己，即使函数也是如此。然而，函数封装以后就是新的函数了，不但不 `eq?` 自己，也不应该，因为其行为真的不同了（尽管只是在违反契约的情况下，并且只在提供了输入值以观察违规行为后）。然而，这意味着程序无法暗中保护自己，因为守护行为可以被观察到。因此，恶意模块有时可以检测它收到的是否是受保护的值，如果是就正常运行，否则就不！

16.7 契约和赋值

我们无疑应该关注契约与赋值之间的相互作用，当契约推迟——固有延时或者以延时方式实现——时更是如此。有两件事值得关注。一是将契约值存储在可变状态中；二是为可变状态编写的契约。

当我们存储契约值时，封装策略确保契约检查正常进行。在每个步骤，契约都会尽可能多地检查现有的值，并创建其余检查的封装值。因此，即使这个封装值被存储在可变状态并在稍后检索以供使用，它仍然包含这些检查，并且当值最终被使用时它们将被执行。

另一个问题是编写可变数据的契约，如box和向量。在这种情况下，我们可能必须为包含契约的整个数据类型创建封装。然后，当数据类型中的值被替换为新值时，执行更新的操作（例如 `set-box!` ）需要从封装中检索契约，将其应用于新值并存储新封装的值。因此，这需要修改数据结构赋值操作符的行为，使其对契约值敏感。然而，赋值不会改变违规行为的发生点：即时契约即时发生，延时契约遇到（非法）输入值时发生。

16.8 契约的组合

我们已经讨论过所有基本数据类型的组合，本节很自然要契约的组合。正如之前讨论的联合和交叉类型一样，我们应该考虑契约的联合和交叉（分别是“或”与“和”）；还应当考虑取反。然而，契约只是表面上类似于类型，所以我们必须根据契约来考虑这些问题，而不是试图将我们从类型学到的意义映射到契约领域。

直接的例子总是简单的。联合契约通过析取组合——事实上，因为是谓词，其结果可以字面上用 `or` 组合——而交叉契约通过合取组合。我们依次调用谓词，进行短路求值（译注，参见后文），最后产生错误或返回契约的值。交叉契约通过合取（ `and` ）组合。而取反契约就是直接调用原始的契约，但对谓词取非（通过 `not` ）。

在延迟、高阶的情况下，契约组合要困难得多。例如，考虑从数到数的函数契约的取反。对它取反到底是什么意思？是否表示该函数不应接受数？或者如果接受了数，它不应该返回数？或两者都要？特别是，我们如何执行这样的契约？例如，如何检查某个函数不接受数——是否期望在给予数时会产生错误？但请考虑用这样的契约封装的恒等函数；因为当给予数（或者其他任何值）时，它显然不会出错，这是否意味着应该等到它产生值，如果它确实产生了数，那么拒绝它？但最糟糕的是，请注意，这意味着我们将在未定义的定义域中运行函数：显然这会破坏程序中的不变量、污染堆栈、或使程序崩溃。

交叉契约要求值通过所有子契约。这意味着高阶值需要重新封装，检查所有定义域子契约以及所有值域子契约。只要一个子契约没有满足，整个交叉（契约）都会失败。

联合契约更加微妙，因为任何一个子契约失败都不直接导致拒绝。相反，它只是意味着这个子契约不再能候选代表契约所封装的值；其他子契约仍然可以候选代表之，只有当没有任何子契约候选时才拒绝值。这意味着联合契约的实现必须在内存中记录哪些子契约通过或失败——而这里内存就意味着需要修改记录。【注释】由于每条子包契约失败时，它将被从候选名单删除，而剩下的会继续执行。当没有候选子契约时，系统必须报告违规行为。错误报告最好要提供导致每个子契约失败的实际值（请记住，这些值可能嵌套在多层函数中）。

> 在类似Racket的多线程语言中，还需要加锁以避免竞争条件。

Racket所实现的契约构造器和组合器对可接受的子契约形式提出了限制。这使得实现既有效率又能提供有用的错误消息。此外，上面讨论的极端情况很少在实践中出现——当然现在如果需要你知道如何实现它们。

16.9 责备

本节回过头讨论报告契约违反的问题。这指的不是打印什么字符串，而是更重要的问题，报告什么。我们将看到，此问题实际上是语义上的考虑。

为了说明这个问题，回想一下上面`d/dx`的定义，假设我们在没有任何契约检查的情况下运行。先假设我们将这个函数应用于完全不合适的`string-append`（它既不读入也不产生数）。这么做只会产生一个值：

```
> (define d/dx-sa (d/dx string-append))
```

（请注意，即使有契约检查，这也会通过，因为函数契约的即时部分认可`string-append`是函数。）接下来假设我们将`d/dx-sa`应用于一个数，这应是正常行为：

```
> (d/dx-sa 10)
string-append: contract violation
  expected: string?
  given: 10.001
```

请注意，错误报告位于`d/dx`函数体的内部。一方面，这完全是合理的：这是`string-append`不正确调用发生的地方。另一方面，错误并非来自`d/dx`，而来自来自提供`string-append`、并声称它是合法的数到数的函数的代码。但问题是，做这件事的代码早已逃之夭夭；它已经不在堆栈中，因此也不在传统错误报告机制的范围内。

这个问题不是`d/dx`所特有的；事实上，大型系统中它很常见。这是因为系统——尤其是图形、网络和其他外部接口的系统——中大量使用回调（callback）：因为对某个实体感兴趣而被注册的函数（或方法），要发某种状态或值的信号时被调用。（在这里，`d/dx`等价于图形层，而`string-append`等价于传给它（并由它存储）的回调。）最终，系统层会调用回调。如

果这会导致错误，那既不是系统层的错误，它收到的回调声明的契约是正确的，也不是回调本身的错误，它应该有合理的用途，只是被错误地提供给函数。相反，错误来源于引入这两者的实体。然而，此时调用栈只包含回调（位于栈顶）和系统（位于其下）——唯一有错的一方不在了。这种类型的错误因此非常难调试。

解决办法是扩展契约系统，纳入责备（blame）【初译注：责任？】的概念。想法是，有效地记录何方导致一对组件汇合在一起，以便如果它们之间发生契约违规，我们可以将失败归因于这一方。请注意，这只是在函数的情况下有意义，但为了一致性，我们以自然的方式将责备扩展到即时契约。

对于函数，请注意有两种可能的失败点：要么它被给予了是错误的值（先验条件），要么是它生成了错误的值（后验条件）。区分这两种情况很重要，因为在前一种情况下，我们应该责备环境——特别是实际参数的表达式——而在后一种情况下（假设参数已经通过），我们应该责备该函数本身。（对即时值的自然延伸，我们只能责备值本身不满足契约，也就是"后验条件"）。

对于契约，我们引入术语正（positive）和负（negative）位置。对于一阶函数，负位置是先验条件，正位置是后验条件。这么看这似乎是不必要的额外术语。但我们很快就会看到，这两个术语具有更一般的含义。

现在将情况推广到契约的参数。之前，即时契约读入一个谓词，而函数契约读入定义域和值域的契约。这点保持不变。不过它们返回的将是函数，此函数有两个参数：正负位置的标签。（这个标签可以是任何合理的数据类型：抽象语法节点、缓冲区偏移量、或其他描述符。简单起见，我们使用字符串。）这样，函数契约将闭包于程序位置标签，以便将来责备非法函数的提供方。

现在由 `guard` 函数负责传入契约调用位置的标签：

```
(define (guard ctc val pos neg) ((ctc pos neg) val))
```

由 `blame` 显示合适的标签（由契约实现传给它）：

```
(define (blame s) (error 'contract s))
```

假设我们像以前一样，保护 `add1` 的使用。正负位置用什么名字有意义呢？正位置是后验条件：这里的任何失败都必须责备 `add1` 的函数体。负位置是先验条件：这里的任何失败都必须责备 `add1` 的参数。因此：

```
(define a1 (guard (function (immediate number?)
                            (immediate number?))
                  add1
                  "add1 body" ;add1函数体
                  "add1 input")) ;add1的输入
```

假设传给`guard`的不是函数，我们会期望在“后验条件”位置出现错误：这并不是后验条件的失败，而是因为，如果调用的不是函数，不能去指责参数。（当然，这表明我们这里扩展了术语“后验条件”，更合理地应该使用术语“正（位置）”。）因为相信`add1`的实现只会返回数，所以我们预计它不可能让后置条件失败。当然，我们期望像`(a1 "x")`这样的表达式触发先验条件错误，可以在`"add1 input"`位置处发出契约错误。相反，如果我们保护的函数违反了后验条件，比如这样，

```
(define bad-a1 (guard (function (immediate number?)
                                (immediate number?))
                      number->string
                      "bad-add1 body"
                      "bad-add1 input"))
```

我们希望责备被归咎于`"bad-add1 body"`。

接下来讨论如何实现这些契约构造函数。对于即时契约，我们说过应把责备归咎于正位置：

```
(define (immediate pred?)
  (lambda (pos neg)
    (lambda (val)
      (if (pred? val) val (blame pos)))))
```

对于函数，我们可能想这么写

```
(define (function dom rng)
  (lambda (pos neg)
    (lambda (val)
      (if (procedure? val)
          (lambda (x) (dom (val (rng x))))
          (blame pos)))))
```

但是这根本不能运作：它违反了契约所预期的签名。这是因为，现在所有契约都期望输入正负位置的标签，也就是`dom`和`rng`不能像上面那样使用。（另一个理由，函数体中用到了`pos`，但完全不含`neg`，尽管已经看到过一些例子，我们认为责备必须归咎于`neg`所绑定的位置。）所以很明显，我们要以某种方式使用`pos`和`neg`实例化的值域和定义域契约，以便它们“知道”和“记住”可能调用非法函数的地方。

最显然的做法是用相同的`dom`和`rng`值实例化这些契约构造函数：

```
(define (function dom rng)
  (lambda (pos neg)
    (let ([dom-c (dom pos neg)]
          [rng-c (rng pos neg)])
      (lambda (val)
        (if (procedure? val)
            (lambda (x) (rng-c (val (dom-c x))))
            (blame pos))))))
```

现在所有签名都匹配了，我们可以运行契约了。但这样做时，返回不太对劲。比如，在我们最简单的违反契约的例子中，返回是

```
> (a1 "x")
contract: add1 body
```

咦？也许我们应该展开 `a1` 的代码，来看看发生了什么。

```
  (a1 "x")
= (guard (function (immediate number?)
                   (immediate number?))
         add1
         "add1 body"
         "add1 input")
= (((function (immediate number?) (immediate number?))
    "add1 body" "add1 input")
   add1)
= (let ([dom-c ((immediate number?) "add1 body" "add1 input")]
        [rng-c ((immediate number?) "add1 body" "add1 input")])
    (lambda (x) (rng-c (add1 (dom-c x)))))
= (let ([dom-c (lambda (val)
                 (if (number? val) val (blame "add1 body")))]
        [rng-c (lambda (val)
                 (if (number? val) val (blame "add1 body")))])
    (lambda (x) (rng-c (add1 (dom-c x)))))
```

可怜的 `add1` ：它都没有获得机会！剩下的唯一责备标签是 `"add1 body"` ，所以只能责备它了。

等下会讨论此问题，先来观察上面的代码，其中没有任何函数契约的踪迹。我们有的只是即时契约，当实际值（如果）发生时进行责备。这与我们之前所说只能观察到即时值完全一致。当然，这只适用于一阶函数；当遇到高阶函数时，这不再成立。

错在哪里？请注意，在 `add1` 函数体中只有绑定到 `rng-c` 的契约应该被责备。相反， `add1` 的输入中应该被责备的是绑定到 `dom-c` 的契约。看起来，在函数契约的定义域位置，正负标签需要……交换。

考虑契约保护的 `d/dx` ，我们会发现情况确实如此。关键的见解是，当调用的函数作为参数时，“外部”成为“内部”，反之亦然。也就是说， `d/dx` 的函数体——处于正位置——调用了被求微分的函数，将这个函数的函数体置于正位置，并将调用者—— `d/dx` 的函数体——置于负位置。因此，在契约的定义域一侧，每次嵌套函数契约都会导致正负位置交换。

值域一侧无需交换。继续考虑 `d/dx` 。它返回的函数代表导数，所以它的输入是数（代表计算导数的点），返回也是数（该点的导数）。这个函数的负位置就是使用微分函数的客户——即先验条件——正位置就是 `d/dx` 本身——即后验条件——因为它负责生成导数。

这样，我们就更正的、正确的函数构造函数的定义：

```
(define (function dom rng)
  (lambda (pos neg)
    (let ([dom-c (dom neg pos)]
          [rng-c (rng pos neg)])
      (lambda (val)
        (if (procedure? val)
            (lambda (x) (rng-c (val (dom-c x))))
            (blame pos))))))
```

练习

> 将此应用于之前的例子，确认得到的责备符合预期。此外，手动展开代码以了解为何。

更进一步，假设我们定义`d/dx`的正位置标签为`"d/dx body"`，负位置标签为`"d/dx input"`。假设我们传给它函数`number->string`（此函数明显无法计算导数），然后将结果应用于`10`：

```
((d/dx (guard (function (immediate number?)
                        (immediate string?))
              number->string
              "n->s body"
              "n->s input"))
 10)
```

这正确地表明，责备应该归咎于将`number->string`作为假定的数函数提供给`d/dx`的表达式——而不是`d/dx`本身。

练习

> 手工计算`d/dx`，将其作用于所有相关的违规情况，并确认由此产生的责备是准确的。如果你将`string->number`传给`d/dx`，附带函数契约指明它将字符串映射到数，会发生什么？如果你在没有契约的情况下传入相同的函数呢？

17 其他调用语义

很久以前，我们讨论过在执行函数调用时替换的问题。现在是时候考虑一些替代方案了。当时，我们只提出了一种方案；其实还有更多选择。要理解这一点，请试着回答这个问题：

下列哪些是相同的？

- `(f x (current-seconds))`
- `(f x (current-seconds))`
- `(f x (current-seconds))`
- `(f x (current-seconds))`

我们将会发现，这段语法可以对应非常不同的运行时行为。比如我们提到过的区别：不同时间求值`(current-seconds)`的不同。另一个不同是，求值的次数有多少（因此`f`运行的次数有多少）。还有一个不同，`x`的值是严格从调用者流向被调用者，还是甚至可能以相反的方向流动！

17.1 惰性调用

先来考虑参数何时规约为值。即，我们是将形参替换为实参的值呢，还是实参表达式本身？如果我们定义

```
(define (sq x) (* x x))
```

然后这样调用

```
(sq (+ 2 3))
```

它是规约为

```
(* 5 5)
```

呢，还是

```
(* (+ 2 3) (+ 2 3))
```

？前者被称为及早（eager）调用，后者则被称为惰性（lazy）调用。【注释】当然，我们不想回到替换模型来定义解释器，但将替换视为设计原则总是有用的。

> 有些人将前者称为严格的（strict）。更加晦涩难解的术语将前者称为调用次序求值（applicative-order evaluation），后者称为正常次序求值（normal-order evaluation）。还有，前者称为传值调用（call-by-value），后者称为传名调用（call-by-name）或传需求调用（call-by-need）。最后这两个术语——传名和传需求——实际技术上有区别，我们将在后文讨论。关于名字的介绍就到这里。

17.1.1 惰性调用示例

惰性这一选择有着辉煌的历史（例如，纯正的λ演算就用它），但淡出了编程实践，(初译意义存疑请校对)问题在于，某些运算符不在调用时对参数求值，而只当需要其值时才求值。例如，考虑定义

```
(define ones (cons 1 ones))
```

在标准Racket中，这显然是有问题的：（左侧的） `ones` 还没有完成定义，我们就（在右侧）尝试对它求值，所以这会导致错误。但是，如果我们不直接对它求值，直到我们真正需要它时，那么这个定义就成立了。因为每次 `rest` 操作都会获得另一个 `ones` ，我们得到了一个无穷链表。

我们略过了很多需要解释的地方。 `cons` 的 `rest` 位置求值得到的是 `ones` 的副本呢，还是原表达式本身呢？换句话说，我们是简单地创建了无限展开的链表，还是创建了实际上循环的链表？

这很大程度上取决于我们的语言是否带有赋值。如果有赋值，那么也许我们可以修改结果链表中的每个单元格，这意味着我们可以观察上述两个实现之间的区别：在展开版本中，修改一个 `first` 不会影响另一个，而在循环版本中，更改一个 `first` 会影响所有其他。因此，在有赋值的语言中，我们可能会倾向与惰性展开，而不是循环数据。

请记住这里的讨论。我们现在无法解决问题；所以我们先仔细检查一下惰性求值，然后回到这个问题。

17.1.2 什么是值？

回到之前的核心高阶函数解释器，我们记得有两种类型的值：数和闭包。要支持惰性求值，我们要问，在函数调用中怎么处理。究竟传入什么？

这似乎很明显：在惰性调用语义中，我们需要传入表达式。但细想就有问题了。表达式中包含标识符名称，【注释】而我们不希望它们被意外地绑定。

> 现在，它们真的是标识符而不是变量，我们马上会发现。

例如，假设我们有

```
(define (f x)
  (lambda (y)
    (+ x y)))
```

这样调用它：

```
((f 3) (+ x 4))
```

思考题

> 这应该返回什么？

显然，应该得到错误，报告x没有被绑定。

现在来逐步分析。第一步调用创建闭包，其中 `x` 绑定到 `3` 。如果接下来将 `y` 绑定到 `(+ x 4)` ，于是得到表达式 `(+ x (+ x 4))` ，而其环境中 `x` 是绑定的。因此我们得到答案 `10` ，而不是错误。

思考题

> 我们这里做了什么微妙的假设吗？

是的，我们有：我们假定 `+` 会对参数求值并返回数作为答案。也许 `+` 也可以是惰性的；我们稍后研究这个问题。不管怎么说，重点不变：如果我们不小心的话，这个错误的表达会得到某种合法的答案，而不是错误。

如果您认为这问题只关于错误的程序，因此可以专门处理（例如，先扫描程序源寻找自由标识符），下面是同一个 `f` 的另一个的用法：

```
(let ([x 5])
  ((f 3) x))
```

思考题

> 这应该返回什么？

正常来说这应该求得 `(+ 3 5)` 的结果（即 `8` ）。但是，如果我们在算术表达式中替换 `x` ，我们会得到 `(+ 3 3)` 。

后面这个例子提示了解决方案的关键所在。在这个例子中，只有当我们用到环境时，问题才会出现；反之如果我们使用替换，一遇到 `let` 就替换函数调用中的 `x` ，结果就符合期望。事实上，请注意，这个观点对前一个例子也适用：如果我们使用替换，那么 `x` 的出现就导致错误。简而言之，我们必须确保基于环境的实现能匹配替换所能做到的。听起来熟悉不！

换种说法，解决方案是将参数表达式与其环境捆绑在一起：即创建闭包。此闭包没有参数，所以它实际上是**thunk**（译注，无参数的**lambda**）。【注释】我们可以使用已有的函数来表示这里的thunk，但是直觉告诉我们，更好的做法是为逻辑上不同的目的使用不同的数据表

示：`closV` 表示用户创建的闭包，用另一种东西表示内部创建的闭包。事实上，正如我们将看到的那样，将它们分开是明智的做法，因为有一个地方我们需要能将它们区分开来。

> 事实上，这表明函数有两个用途：用值替换名称，推迟替换。`let` 只有前一个功能而没有后一个；thunk只有后一个功能而没有前一个。前文已经明确，前者本身是有意义的；本节表明后者也是如此。

总结来说，现在我们新的值的集合是：

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)]
  [suspendV (body : ExprC) (env : Env)])
```

前两个变体完全不变；第三个是新的，正如我们所讨论的，它实际上是一个无参数的子程序，正如其类型所暗示的那样。

17.1.3 何时求值？

回头来讨论算术表达式。对 `(+ 1 2)` 求值时，惰性调用的解释器可以返回好几种东西，包括 `(suspendV (+ 1 2) mt-env)` 。【注释】通过这种方式，挂起的计算可以串上挂起的计算，在极限情况下，任何程序都会立即返回“答案”：表示暂停计算的thunk。

> 这里放上 `mt-env`（译注，空白环境）是合理的，因为就算 `(+ 1 2)` 表达式存在与非空的环境中，它其中也不包含自有标识符，因此不需要任何环境的绑定。

显然，必须有啥东西来强制解除暂停。（当然，解除暂停的意思是，在存储下来的环境中对主体求值。）这种解除表达式暂停的位置称为严格点（**strictness point**）。最明显的严格点是交互式环境的打印，因为如果用户使用交互环境显然是希望看到答案。我们用 `strict` 子程序表示解除暂停：

```
(define (strict [v : Value]) : Value
  (type-case Value v
    [numV (n) v]
    [closV (a b e) v]
    [suspendV (b e) (strict (interp b e))]))
```

这里返回的 `Value` 保证不是 `suspendV` 。我们可以假设打印程序会对被求值的表达式调用 `strict` ，以获得要打印的值。

思考题

> 如果使用闭包来表示暂停计算，后果是啥？

这个 `strict` 定义依赖于区分延迟计算的能力——哪些是内部构建的闭包，哪些是用户定义的闭包。如果我们将两者混为一谈，那么这里就不得不猜测如何处理零参数闭包。如果没有进一步处理它们，我们可能会错误地得到报错（例如， `+` 可能会得到thunk而不是其中的数

值）。如果进一步处理，我们可能会意外地过早调用用户定义的thunk。总之，对于thunk我们需要一个标志，告诉我们它们是内部的还是用户定义的。为了清晰起见，我们的解释器使用独立的变体。

接下来讨论`strict`和解释器之间的互动。不幸的是，按我们原来的定义，这将导致无限循环。要解释加法，需要先创建暂停，`strict`会试图解除暂停，这需要用解释器来解释加法，而这又……显然，我们不能让每个表达式都简单的暂停计算；相反，我们只暂停函数调用。这不会使语言变得荒谬，又足以让我们拥有惰性求值的强大力量。

17.1.4 解释器

照例，我们将分步定义解释器。

```
<lazy-interp> ::=

    (define (interp [expr : ExprC] [env : Env]) : Value
      (type-case ExprC expr
        <lazy-numC-case>
        <lazy-idC-case>
        <lazy-plusC/multC-case>
        <lazy-appC-case>
        <lazy-lamC-case>))
```

数很容易：它们已经是值了，所以没必要暂停它们：

```
<lazy-numC-case> ::=

    [numC (n) (numV n)]
```

闭包同样保持不变：

```
<lazy-lamC-case> ::=

    [lamC (a b) (closV a b env)]
```

标识符应该返回它们所绑定的内容：

```
<lazy-idC-case> ::=

    [idC (n) (lookup n env)]
```

算术表达式的参数通常被定义为严格点，不然的话我们会不得不在其他地方实现实际的算术运算：

```
<lazy-plusC/multC-case> ::=

    [plusC (l r) (num+ (strict (interp l env))
                       (strict (interp r env)))]
    [multC (l r) (num* (strict (interp l env))
                       (strict (interp r env)))]
```

最后我们要处理函数调用。在这里，我们不再对参数求值，而是将其暂停。然而，函数位置必须是严格点，否则我们不知道要调用什么函数，也就不知道如何继续计算：

```
<lazy-appC-case> ::=

    [appC (f a) (local ([define f-value (strict (interp f env))])
                  (interp (closV-body f-value)
                          (extend-env (bind (closV-arg f-value)
                                            (suspendV a env))
                                      (closV-env f-value))))]
```

这就行了！添加一种新的答案、插入一些 `strict` 、并在函数调用参数位置用 `suspendV` 替换 `interp` ，我们就将及早调用解释器转换成了惰性调用了。然而，这个小小的变化对我们编写的程序有着巨大的影响！要更全面地了解这种影响，请学习Haskell或Racket中的 `#lang lazy` 语言。

练习

> 如果我们把标识符子句替换为 `(strict (lookup n env))` （即对查找标识符的结果调用 `strict` ），会对语言产生什么影响？请考虑更丰富的语言的情况，比如包含数据结构的情况。

练习

> 编写一些程序，它们在惰性求值下会给出和及早求值不同的结果（在两种情况下，同样的程序给出不同的结果）。请给出有意义的差异，一个返回 `suspendV` 而另一个不不算。比如说，一个会终止而另一个不会，或者一个会产生错误而另一个不会？

练习

> 调整两个解释器，让它们记录求得答案的步数。对于在两种求值策略下产生相同答案的程序，一个策略是否总是比另一个需要更多步骤？

17.1.5 惰性和赋值

惰性求值的优点之一是它会延迟执行。通常这是好事：它使我们能够构建无限的数据结构，还能避免不必要的计算。不幸的是，它也改变了计算发生的时间，尤其是表达式求值的相对时间，这取决于何时遇到严格点。结果是，程序员基本无法预测计算的顺序。当表达式执行赋值操作时，这显然是个问题，因为现在要预测程序将计算出什么值变得非常困难（相对及早求值来说）。

这导致了，所有惰性语言的核心中都不支持赋值。在Haskell中，赋值和其他状态操作都通过**monad**（单子）和**arrow**（箭头）等多种机制，引入了（严格）序列化代码的能力后再引入；这种顺序性对于能够预测执行顺序以及操作结果至关重要。如果程序结构良好，这些依赖关系的数量应该很小；此外，Haskell类型系统试图在类型本身中反映这些操作，因此程序员可以更轻松地推理其效果。

17.1.6 缓存计算结果

既然已经得出结论，惰性计算必须不包含赋值，我们观察到一个令人愉快的结果（能不能称其为副作用呢？）：给定固定的环境，同一表达式总会产生相同的答案。其结果是，当表达式第一次被严格求值时，运行时系统可以缓存其值，并在随后计算它时返回这个缓存值。当然，这种缓存（这是记忆化(memoization)的一种形式）只有当表达式每次返回相同的值时才成立，这正是我们所假设的。实际上，编译器和运行时系统可以积极地在程序的不同部分中使用相同的表达式，并且如果其环境的相关部分相同，则合并求值。每当需要被暂停的计算时都求值的策略称为传名调用；将结果缓存起来，则称为传需求调用。

17.2 响应式调用

来考虑这个表达式`(current-seconds)`。求值时，它返回一个数，代表当前时间。例如，

```
> (current-seconds)
1353030630
```

但就算我们盯着这个值，它已经过时了！它表示函数调用发生的时间，而不会不保持当前秒数。

17.2.1 动机样例：计时器

假设我们要实现一个计时器，记录经过的时间。理想情况下，我们想这样写：

```
(let ([start (current-seconds)])
  (- (current-seconds)
     start))
```

在JavaScript中就是：

```
d = new Date();
start = d.getTime();
current = d.getTime();
elapsed = current - start;
```

在大多数机器上，此Racket表达式，或JavaScript中 `elapsed` 的值将被求得为 `0` ，或着某个非常小的数字。这是因为这些程序代表了经过时间的一次度量：即第二次调用获取当前时间子程序时的时间。这样我们拿到一个瞬间的时间值，而不是实际的计时器。

在大多数语言中，要构建真正的计时器，我们必须创建某种计时器对象的实例，然后设置回调。每当时钟滴答时，计时器对象——这里代表操作系统——都会调用回调函数。然后回调负责更新系统其余部分的值，祈求这能全局并一致地完成。但是，回调函数无法通过返回值来实现这点，因为它会返回回到操作系统，而操作系统无法预知我们的应用程序，也不关心；因此，回调只能通过赋值来执行其行为。例如在JavaScript中：

```
var timerID = null;
var elapsedTime = 0;

function doEverySecond() {
  elapsedTime += 1;
  document.getElementById('curTime').innerHTML = elapsedTime; }
function startTimer() {
  timerId = setInterval(doEverySecond, 1000); }
```

假设这里的HTML页面id为 `curTime` ，并且 `onload` 或其他回调会调用 `startTimer` 。

要避免这种意大利面风格的代码，一种替代方案是应用程序反复向操作系统轮询当前时间。然而：

- 过于频繁地调用会浪费资源，而调用过于不频繁则会导致错误的值。不过，要以恰当的频率进行调用，我们需要先有一个计时器信号！
- 尽管可以为诸如定时器之类的常规事件创建这样的轮询循环，但对于诸如用户输入等不可预知的行为（其频率通常不能被预测）的来说，这是不可能的。
- 除此之外，编写这样的循环会污染程序的结构，迫使开发人员承担额外的负担。

然而，基于回调的解决方案展示了控制的倒置（inversion of control）。现在，操作系统负责调用（从而进入）应用程序，而不是应用程序调用操作系统（所提供的功能）。理论上响应行为应该在深度嵌套于显示表达式的内部，但它实际上位于顶层，其值会驱动其他计算。这么做的根本原因在于，控制掌握在世界而不是程序手中，所以外部刺激而非内在的程序表达式决定了程序何时运行以及如何运行。

17.2.2 回调的类型是四字母单词

这种模式的特征（或者说签名）体现在类型中。由于操作系统对程序的值并不知情，所以回调通常没有返回类型，或者只返回通用的状态指示值，而不是特定于应用程序的值。因此，在静态类型语言中，它们的类型通常是四个字母的单词。 例如，下面是Java中某GUI库的片段：

```
interface ChangeListener extends EventListener {
  void stateChanged(ChangeEvent e) { ... } }

interface ActionListener extends EventListener {
  void actionPerformed(ActionEvent e) { ... } }

interface MouseListener extends EventListener {
  void mouseClicked(MouseEvent e) { ... }
  void mouseEntered(MouseEvent e) { ... } }
```

OCaml中是这样：

```
mainLoop : unit -> unit
closeTk : unit -> unit

destroy : 'a Widget.widget -> unit
update : unit -> unit

pack : ... -> 'd Widget.widget list -> unit
grid : ... -> 'b Widget.widget list -> unit
```

在Haskell中，这四个字母中包含一个额外的空格：

```
select :: Selecting w => Event w (IO ())
mouse :: Reactive w => Event w (EventMouse -> IO ())
keyboard :: Reactive w => Event w (EventKey -> IO ())
resize :: Reactive w => Event w (IO ())
focus :: Reactive w => Event w (Bool -> IO ())
activate :: Reactive w => Event w (Bool -> IO ())
```

诸如此类。在所有这些情况下，类似“void”类型的存在清楚地表明这些函数不会返回任何有意义的值，所以它们唯一的目的必须是修改贮存或者具有其他副作用。这也意味着复杂的组合手段（例如表达式的嵌套）是不可能的：void类型语句唯一的组合操作是顺序执行。因此这些类型表明我们将被迫放弃编写嵌套表达式。

当然，通过我们之前对Web编程的讨论，读者熟知这个问题。由于没有状态，服务器上有这个问题；由于单线程，客户端上也有这个问题。至少在服务器上，我们能够用continuation解决这个问题。但是，不是所有的语言都支持continuation，并且实现continuation也会很繁琐。此外，设置合适的continuation作为回调来传递可能会非常棘手。因此，我们将探索另一种解决方案。

17.2.3 替代方案：响应式语言

考虑DrRacket中的FrTime（发音为“Father Time”）语言。【注释】如果我们在交互窗口中运行下面的表达式，我们仍然得到0或者非常小的正数：

```
(let ([start (current-seconds)])
  (- (current-seconds)
     start))
```

> 在DrRacket v5.3中，必须从“语言/Language”菜单中选择该语言；只写`#lang frtime`不会提供想要的交互窗口行为。

事实上，我们可以尝试其他几种表达式，看上去FrTime似乎与传统的Racket完全一样。

但是，它还绑定了额外一些标识符。例如，有一个值绑定到`seconds`。如果我们将其输入交互窗口的提示符，结果非常有意思！首先我们看到`1353030630`，然后一秒后`1353030631`，再一秒`1353030632`，诸如此类。这种值被称为行为（behavior）：随时间变化的值。但是我们没有编写任何回调或其他代码将其值保持最新。

行为可以用于计算。 例如可以这么写`(- seconds seconds)`，并且它总是计算为`0`。请在交互提示符中尝试更多表达式：

```
(add1 seconds)
(modulo seconds 10)
(build-list (modulo seconds 10) identity)
(build-list (add1 (modulo seconds 10)) identity)
```

正如你所看到的，行为是“粘性的”：如果任何子表达式是行为，包含它的表达式也是。

基于这里的求值模型，每当`seconds`更新，整个应用程序重新求值：因此，即使我们写了看似简单的表达式，不包含任何明确的循环控制，程序仍然会“循环”。换句话说，最早我们探索的调用语义，其中参数被求值一次，接下来的调用语义中，参数可能被求值零次，现在这个调用语义会根据需要对参数以及与它们对应的整个函数进行多次求值。因此，表达式“内部”的响应式值不再需要被带到“外部”；相反，它们可以嵌套在表达式中，为程序员提供更自然的表达方式。这种评估方式称为数据流（dataflow）或函数响应式（functional reactive）编程。

> 历史上，数据流一般指的是语言具有一阶函数，而函数响应式语言支持高阶函数。

FrTime实现了我们所说的透明响应式，即程序员可以在程序求值的任意位置插入响应行为，而无需对其上下文进行任何语法修改。这么做的优点是，现有程序中很易于加入响应式，但这也使求值模型更加复杂，程序员估计复杂度也变难了。在其他语言中，程序员需要通过适当的原语明确地引入行为，不那么方便，但可预测性更强。FrTime的姊妹语言Flapjax是JavaScript的扩展，同时支持这两种模式。

> 参见Flapjax网站。

17.2.4 实现透明响应式

要使现有语言实现透明响应式，我们必须（自然地）改变函数调用的语义。分两步来做。首先将响应式函数调用改写成更复杂的形式，然后我们将展示这种更复杂的形式支持响应式更新。

17.2.4.1 数据流图的构建

使函数调用具有响应性的本质很容易通过去语法糖来解释。假设我们已经定义了新的构造器 `behavior`。该构造器的输入为一个thunk，表示每次参数更新时要执行的计算，以及表达式所依赖的所有的值。构造器的返回值存储行为的当前值。那么 `(f x y)` 这样的表达式就展开为

```
(if (or (behavior? x) (behavior? y))
    (behavior (λ () (f (current-value x) (current-value y))) x y)
    (f x y))
```

其中我们假设，如果输入是常数而非行为，那么 `current-value` 的行为就是恒等函数。

来看一下使用上述定义的两个例子。考虑两个参数都不是行为的简单情况，例如 `(+ 3 4)`。去语法糖得到

```
(if (or (behavior? 3) (behavior? 4))
    (behavior (λ () (+ (current-value 3) (current-value 4))) 3 4)
    (+ 3 4))
```

由于 `3` 和 `4` 都是数而非行为，这就规约为 `(+ 3 4)`，正是我们想要的。这反映了一个重要的原则：当没有行为出现时，程序的行为完全等同于与非响应式语言版本。

如果计算 `(+ 1 seconds)`，展开为

```
(if (or (behavior? 1) (behavior? seconds))
    (behavior (λ () (+ (current-value 1) (current-value seconds))) 1 seconds)
    (+ 1 seconds))
```

由于 `seconds` 是行为，这规约为

```
(behavior (λ () (+ (current-value 1) (current-value seconds))) 1 seconds)
```

如果其他表达式依赖于此，现在它们都会看到其参数也是行为，于是该属性如我们之前所论证的那样是“粘性的”。

练习

> 上述去语法糖是否依赖于及早求值？如果有的话，是以什么方式？

17.2.4.2 数据流图的更新

当然，仅仅构建行为值是不够的。这里关键的附加信息位于 `behavior` 的参数中。语言会过滤掉那些本身是行为的参数（例如前述的 `seconds`），并将新行为注册为取决于现有行为的行为。这个注册过程创建了行为表达式的依赖关系图，称为数据流图(dataflow graph)（因为它反映了数据流动所需的路径）。

如果程序求值得到的不是行为，那么只能是答案，并且这也不会创建图表。但是，如果存在行为依赖，那么求值不会产生传统的答案，而会产生行为值，并且会记录其依赖。（实践中，有必要记录下哪些原始行为实际地被用到，以避免不必要地对程序中没有引用的其他原始行为求值）。总之，程序执行会生成数据流图。因此，我们需要的不是新的、专门的语言求值器；而是要将图形构建语义嵌入到传统求值器中。

现在可以运行数据流传播算法了。每当原始行为发生变化时，该算法会调用其存储的thunk，获取新值，存储之，然后发信号给依赖于它的所有行为。例如，如果 `seconds` 更新，它会通知对应表达式 `(+ 1 seconds)` 的行为。后者于是对其thunk求值，即 `(λ () (+ (current-value 1) (current-value seconds)))` 。这会对 `seconds` 的最新值加 `1` ，将其作为该行为的的新值——正如我们所期望的那样。

17.2.4.3 求值顺序

上面对图更新的讨论过于简单了。考虑以下程序：

```
(> (add1 seconds)
   seconds)
```

这个程序里有一个原始行为， `seconds` ，构造了两个新行为：分别是 `(add1 seconds)` 和整个表达式。

我们期望这个表达永远计算为真。但是，当 `seconds` 更新时，取决于处理更新的顺序，可能会在更新 `(add1 seconds)` 之前更新整个表达式。假设 `seconds` 的旧值是 `100` ，所以新值是 `101` 。但是， `(add1 seconds)` 的节点仍然存储了其旧值（因为它尚未更新），所以它的值还是 `(add1 100)` 即 `101` 。这意味着 `>` 会比较 `101` 与 `1` （译注，此处应为 `101` ），得到假，于是这个表达式返回了其静态描述不可能产生的值。这种情况被称为毛刺（glitch）。

避免上面例子所描述的毛刺的方案很简单（而且可以证明这么做足够了）。就是对节点拓扑排序。每个节点只在它所依赖的节点更新后才被处理，因此不存在查看过时或不一致的值的危险。

在图中出现循环时问题变得难了。在这种情况下，我们需要特殊的递归算子来为循环行为提供初始值。这样做就打破了循环依赖关系，将求值简化为已定义的过程。

关于数据流语言的求值还有很多可以讨论的内容，例如条件的处理、还有离散和流式（stream-like）行为对偶的概念。 我希望你会去阅读响应式语言的文献，以便更多地了解这些主题。

练习

> 之前我们提到过一个Haskell库。不过，公平地说，我们展示的响应式解决方案是用Haskell来阐述的，因为惰性求值更容易支持这种求值形式。
>
> 用惰性求值实现响应式。

17.3 回溯调用

同一个调用可能发生多次的另一个原因是，它是搜索树的一部分。这类语言的调用语义试图去满足搜索；如果成功，则返回成功的信息，但如果失败，则会重试调用以期成功。当然，这假定程序是按照可以尝试的选项编写的，直到搜索成功。因此，具有回溯调用语义的语言的核心操作是逻辑析取（**disjunction**，或）。出于各种原因，此类语言还支持逻辑合取（**conjunction**，与），其中一个原因是，实现逻辑非会有问题，所以通常的布尔代数规则并不适用。

17.3.1 通过搜索获得满足

要描述回溯搜索问题，最简单的方式是基于简单二进制值目标。（TODO:不满意这一段翻译，校对请修改）对布尔变量来说，即使只是寻找满足命题公式的解，从性能的角度来看计算上是有挑战性的，并且这在各种现实世界问题中非常重要。【注释1】然而，我们只讨论这个问题的简化版本，只使用布尔常量而非变量，这样我们只需要确定公式的真实性就可以了。不那么有趣，但它会帮助我们建立一般的、实际上有趣的情况。【注释2】

> 参见“SAT求解器”的众多用途。
>
> 对于这种特殊情况，制定真值表就可以了，但对一般情况这不起作用。

假设我们的输入是包含析取、合取和表示真假的常量的公式。目标是判定公式本身求值为真还是假。我们希望尽量减少计算量，当发现答案——无论哪一个——我们都希望尽快将其返回到依赖于它的上下文。例如，如果在计算某个合取过程中发现某个项为假，我们希望整个（合取）项立即得假——条件表达式求值中的短路求值概念。而且，我们也希望这种做法能泛化到调用堆栈中：如果子表达式求得真或假，并且这可以决定包含表达式的值了，那么它也应该快速地“通知所在堆栈”。

因此，一般来说，每个计算都应该再带两个容器参数：一个用来报告前术语为真（如果发现是真），另一个报告为假（如果发现是假）。为了避免未决函数调用等问题的复杂性，我们认为这两个参数的值都应是**continuation**，以便该值尽可能快地回到正确的上下文中，而不通过中间级别的求值，那些步骤和结果无关。

这些continuation目前无需传递有意义的值：所需的信息由哪一个延续被调用提供（一比特的信息，代表真或假）。因为默认情况下continuation需要有个参数，我们用一个符号来表示已知的内容。

最容易的值是真和假本身。之前说过，所有表达式都会读入两个continuation，称为成功和失败continuation，如果式子具有明确的值，则调用其中一个。因此，真值调用成功continuation，而假值调用失败：

```
(define (truth t1 t2) (t1 'yes))
(define (falsity t1 t2) (t2 'no))
```

现在我们来讨论析取。为了简单起见，我们将讨论其双目版本。和所有计算一样，两个回溯搜索的析取必须接受成功和失败continuation。

```
<try-or-bt> ::=

    (define (try-or t1 t2)
      (lambda (success failure)
        <try-or-bt-body>))
```

从概念上讲，最简单的方式是创建两个局部continuation，称之为 `pass` 和 `fail` ，并传递给 `t1` 求值。如果t1（或者递归地，它的某个子计算）成功，控制将返回到创建 `pass` 的上下文；如果失败，则返回到 `fail` 。

如果控制返回到 `pass` ，我们就知道第一个子表达式成功了。但是因为对于析取，这就够了，所以我们现在可以将控制交给continuation `success` 。因此任何对 `pass` 的调用都应该立即触发 `success` 。

反之，假设 `t1` 失败。那我们应该试试 `t2` 。因此 `fail` 应被定义在序列操作中，接下来要尝试 `t2` ；如果 `t1` 成功，控制将不会以这种方式返回。接下来尝试 `t2` 时，不必担心 `pass` 和 `fail` ：在 `t1` 失败后，整个析取的成功和失败等同于 `t2` （尾位置的一种形式）的成功和失败，因此它的成功和失败continuation与整个表达式的相同。于是，我们获得：

```
<try-or-bt-body> ::=

    (success (let/cc pass
               (begin
                 (let/cc fail
                   (t1 pass fail))
                 (t2 success failure))))
```

因此，如果 `t1` 成功，则控制返回到创建 `pass` 的上下文，即调用 `success` 。如果 `t1` 成功（译注，应为失败），则控制返回到创建 `fail` 的continuation，也就是序列中的下一个语句， `t2` 。

根据对称推理，我们可以得到对偶的 `try-and` 程序：

```
(define (try-and t1 t2)
  (lambda (success failure)
    (failure (let/cc fail
               (begin
                 (let/cc pass
                   (t1 pass fail))
                 (t2 success failure))))))
```

为了方便测试，我们可以编写封装函数，将这些基于continuation的回复转换为简单的值：

```
(define (run t)
  (let/cc escape
    (t (lambda (v) (escape 'yes))
       (lambda (v) (escape 'no)))))
```

然后以此创建测试案例，从

```
(test (run (try-or falsity falsity)) 'no)
```

到

```
(test (run (try-or (try-and (try-or falsity truth) (try-or truth falsity))
                   (try-and truth (try-and falsity truth)))) 'yes)
```